F77L

FORTRAN 77 LANGUAGE SYSTEM FOR THE PERSONAL COMPUTER

Reference Manual

Lahey
Computer Systems Inc.

PROGRAM LICENSE AGREEMENT

PLEASE READ THE FOLLOWING TERMS AND CONDITIONS BEFORE USING THIS PROGRAM. USE OF THE PROGRAM INDICATES YOUR ACCEPTANCE OF THESE TERMS AND CONDITIONS. IF YOU DO NOT AGREE WITH THEM, RETURN THE PROGRAM ALONG WITH PROOF OF PURCHASE AND YOUR MONEY WILL BE REFUNDED.

Lahey Computer Systems, Inc. (LCS) provides this program and licenses its use. You assume responsibility for the selection of the program to achieve your intended results, and for the installation, use and results obtained from the program.

LICENSE

You may:

1. use the program on any machine or machines in your possession, but you may not have a copy on more than one machine at any given time.
2. copy the program into any machine-readable or printed form for backup or modifications purposes in support of your use of the program.
3. modify the program and or merge into another program for your use (any portion of the program merged into another program will continue to be subject to the terms and conditions of this Agreement);
4. transfer the program and license to another party if the other party agrees to accept the terms and conditions of this Agreement. If you transfer the program, you must at the same time either transfer all copies whether in printed or machine-readable form to the same party or destroy any copies not transferred: this includes all modifications and portions of the program contained or merged into other programs. When transferring the license to another party, please inform LCS as to the name of the new registered owner.
5. create User Programs using the program. Such User Programs may include copies of code derived from the libraries and/or bound from the object libraries, and/or modifications of these libraries. You may ship the error file, *.EER, with your programs. User Programs do not have to include our copyright notices, do not have to be destroyed at the termination of this Agreement, and are not subject to any usage and/or transfer restrictions.

You must reproduce and include copyright notices on any copy, modifications or portion merged into another program, with the exception of User Programs. You may not use, copy, modify, or transfer the program, or any copy modification or merged portion, in whole or in part, except as expressly provided for in this license. If you transfer possession of any copy, modifications or merged portion of the program to another party, except as expressly provided for in this license, your license is automatically terminated. You may not reverse engineer, decompile, disassemble, or create derivative works based on the software.

TERM

The license is effective until terminated. You may terminate it at any other time by destroying the program together with all copies, modifications and merged portions in any form. It will also terminate upon conditions set forth elsewhere in this Agreement or if you fail to comply with any term or condition of this Agreement. You agree upon such termination to destroy the program together with all copies, modifications and merged portions in any form with the exception of User Programs.

LIMITED WARRANTY

With respect to the physical diskette and physical documentation enclosed herein, Lahey Computer Systems, Inc. (LCS), warrants the same to be free of defects in materials and workmanship for a period of 30 days from the date of purchase. In the event of notification within the warranty period of defects in material or workmanship, LCS will replace the defective diskette or documentation. The remedy for breach of this warranty shall be limited to replacement and shall not encompass any other damages, including but not limited to loss of profit, special incidental, consequential, or other similar claims.

Lahey Computer Systems, Inc. specifically disclaims all other warranties, expressed or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose with respect to defects in the diskette and documentation, and the program license granted herein, in particular, and without limiting operation of the program license with respect to any particular application, use, or purpose.

GENERAL

Refer to the LCS Customer Relations Policy in this manual for a complete description of LCS's product update, technical support, problem resolution and product return policies. Lahey Computer Systems, Inc. does not warrant that operation of the program will be uninterrupted or error free.

You may not sublicense, assign or transfer the license or program except as expressly provided in this Agreement. Any attempt otherwise to sublicense, assign or transfer any of the rights, duties or obligations hereunder is void.

This Agreement will be governed by the laws of the State of Nevada.

You acknowledge that you have read this agreement, understand it and agree to be bound by its terms and conditions. You further agree that it is the complete and exclusive agreement between us which supersedes any proposal or prior agreement, oral or written, and any other communications between us relating to the subject matter of this Agreement.

If you have any questions concerning this Agreement, please contact Lahey Computer Systems, Inc.

F77L

FORTRAN LANGUAGE SYSTEM REFERENCE MANUAL

Revision E
August 1989

SYSTEM REQUIREMENTS

To use the F77L Language System you must meet the following hardware and software requirements.

Hardware Requirements:

- An 8086-, 8088-, 80286-, or 80386-based IBM, Compaq or fully compatible computer;
- An 80386-based computer with a Weitek 1167 or 3167 (Abacus) Math Chip is required to use the /K Weitek Option; and
- At least 256KB of RAM for the compiler and 384KB for the Lahey Blackbeard Editor.

Software Requirements:

- MS-DOS or PC-DOS Versions 2.0 or higher; and
- A virtual-8086 device driver, e.g. "CEMM.SYS" for Compaq 80386s, is required for the Weitek Option. Refer to the "Weitek Considerations" section in the READ.ME File for up-to-date information on specific drivers.

Installation:

To install F77L see **Appendix A – "Installation and Operation"** in the Reference Manual. Please be sure to read the file "READ.ME" on the F77L Disk for the latest information on F77L.

COPYRIGHT

DISCLAIMER

Lahey Computer Systems, Inc. reserves the right to revise its software and publications with no obligation of Lahey Computer Systems, Inc. to notify any person or any organization of such revision. In no event shall Lahey Computer Systems, Inc. be liable for any loss of profit or any other commercial damage, including but not limited to special, consequential or other damages.

TRADEMARKS

- Compaq is a trademark of Compaq Computer Corp.
- DEC VAX is a trademark of Digital Equipment Corp.
- Hercules is a trademark of Hercules Computer Technology.
- IBM, PC-DOS, and IBM VS are trademarks of International Business Machines Corp.
- INTEL, 8086, 8088, 80286, and 80386 are trademarks of Intel, Inc.
- Lattice C is a trademark of Lattice, Inc.
- Microsoft, MS-DOS, Microsoft FORTRAN, Microsoft C, CodeView, OS/2, Microsoft Make, Microsoft Macro Assembler, and Microsoft Link are trademarks of Microsoft, Inc.
- OPTLINK is a trademark of SLR Systems.
- PLINK86 Plus is a trademark of Phoenix Technologies, Inc.
- Turbo C and TLINK are trademarks of Borland International.
- Weitek 1176 & 3167 Abacus are trademarks of Weitek, Corp.
- UNIX is a trademark of Bell Laboratories.

Lahey Computer Systems, Inc.
P.O. Box 6091
Incline Village, NV 89450-6091
(702) 831-2500

CONTENTS

DATA TYPES and CONSTANTS 2

ARRAYS 3

EXPRESSIONS 4

COMPILER-DIRECTIVE STATEMENTS 5

SPECIFICATION and DATA STATEMENTS 6

ASSIGNMENT STATEMENTS 7

CONTROL STATEMENTS 8

INPUT/OUTPUT STATEMENTS 9

EXECUTABLE PROGRAMS 10

INTRINSIC FUNCTIONS 11

SYSTEM SUBROUTINES 12

INSTALLATION and OPERATION A

PROGRAMMING HINTS B

LAHEY BLACKBEARD EDITOR C

LAHEY MAKE BY OPUS SOFTWARE D

LAHEY LIBRARY MANAGER E

LAHEY OPTLINK LINKER BY SLR SYSTEMS F

INTRODUCTION

I INTRODUCTION

This section defines the relationship between Lahey Computer Systems, Inc. and its users. The following pages contain information on these topics:

- **Lahey Customer Relations**
- **How to Contact Lahey**
- **Product Version Numbers and Update Policies**
- **Technical Support Services**
- **How to Report Problems**
- **Lahey's Product Return Policies**

I.1 Lahey Customer Relations

At Lahey Computer Systems, Inc. (Lahey) we take pride in the relationship we have with our customers. This relationship is based on maintaining good communications with our users. This communication includes providing quality technical support (resulting in problem resolution in days or weeks, not months), having an electronic bulletin board system and sending out newsletters, product brochures and new release announcements. In addition, we listen carefully to our users' comments and suggestions and often incorporate their ideas into our products.

To make it convenient for our users, we have several ways to communicate with us:

TEL:	**(702) 831-2500**
FAX:	**(702) 831-8123**
TELEX:	**9102401256**
BBS:	**(702) 831-8023**
UUNET	**uunet!lahey!support or** **uunet!lahey!sales**
US MAIL:	**PO Box 6091** **Incline Village, NV 89450-6091**

To establish this communication, please complete and mail the enclosed **Lahey Product Registration Card**. If you move or transfer a Lahey product's ownership, please fill out and return the **Change of Address/Ownership Card** provided. By using our software, you agree to observe the terms of the Lahey License Agreement. Lahey in turn agrees to provide technical support and to abide by the support and return policies explained on the following pages.

I.2 Version Numbers & Update Policies

Associated with each release of a Lahey product is a version number, found on the software disk label, which has three numeric fields, i.e., "1.23". These fields are changed for each release to indicate qualitatively what has been modified. The meaning of changes to these fields is as follows:

> A "major release", designated by a change to the first field, occurs when enhancements are made to the software and/or the user manual has been revised. The major release succeeding Version "1.23" would be numbered Version "2.00".
>
> A "minor release" which changes the second field, i.e., "1.23" to "1.30", reflects some changes to the software along with an addendum to the manual.
>
> A change to the last field indicates an "update". Version "1.23" would be followed by "1.24". Changes are documented in the "READ.ME" file on the product diskette.

If a major or minor release is made, Lahey stops fixing problems in all earlier versions 60 days after the new release date. If the new release is an update, you continue to receive problem resolution for earlier versions at no cost.

Problem resolution is defined as a technical support inquiry that requires a new version to correct the problem. If your technical support inquiry requires problem resolution, and either of the first two numbers of the version number have been changed for more than 60 days, then you will need to purchase the new release.

The update fee for a new release depends upon how much of the software has been changed since the previous release. Lahey Computer Systems, Inc. reserves the right to change its products, its support of Lahey-compatible products, its documentation and update fees for any release without prior notification.

I.2.1 Lahey-Supported Compatible Products

Lahey recognizes that support for all future versions of Lahey-compatible products with Lahey-produced interfaces and libraries (as identified in this user manual) cannot always be warranted. At the same time Lahey recognizes that supporting these Lahey-compatible products increases the quality and usability of its own products. The following policy is intended to help resolve any potential conflicts between the goal of supporting future releases of Lahey-supported compatible products and the possible necessity of discontinuing support.

Lahey's goal is to support new versions of compatible products within 120 days of their release. Lahey may at any time elect to discontinue support for a compatible product for the following reasons: lack of Lahey customer interest, technical problems with the support interface which could have repercussions in Lahey's own products, and lack of cooperation from the compatible product's developers. Lahey suggests that its users continue to use proven interfaces to compatible products until Lahey announces full compatibility with a new release.

I.3 Technical Support Services

Lahey provides technical support at no cost to users. This support is limited to software problems and does not include tutoring in how to program in FORTRAN or how to use DOS. The first level of technical support services are inquiries than can be answered or corrected and do not require a new version of the software. All registered users receive this first level of technical support regardless of the version of their Lahey software. The following communication methods are supported:

TEL:	**(702) 831-2500 — (support hours: 9:30 AM to 3:30 PM Pacific Time)**
FAX:	**(702) 831-8123 — (available 24 hours)**
TELEX:	**9102401256 — (available 24 hours)**
BBS:	**(702) 831-8023 — (available 22 hours)**
UUNET	**uunet!lahey!support or uunet!lahey!sales — (available 24 hours)**

The BBS is available to registered users 22 hours a day, 5 days a week and 24 hours a day on weekends. The BBS is off-line from 3:30 to 5:30 PM Pacific Time, Monday through Friday, for system maintenance. The BBS enables Lahey to respond quickly to your inquiries and to provide fixes to compiler problems. New Lahey product announcements, lists of resolved problems, lists of Lahey-compatible software vendors and user-written utilities are all maintained on the BBS. Users receive faster service if they take advantage of the BBS.

I.4 How To Report Problems

I.4.1 How Lahey Resolves Problems

Contact us if you have a problem with any Lahey product. Many users' questions can be easily handled over the phone. We can also correct certain problems either by having you download the ".FIX" file using the BBS, sending a patch via FAX or TELEX, or by dictating modifications to the ".FIX" file over the phone. Use an ASCII editor to add the FAXed, TELEXed or phone-dictated patch to the ".FIX" file. Other problems may be solved by shipping you a diskette at no cost. Use of this ".FIX" file to resolve problems is unique to Lahey Language Systems.

I.4.2 Methods Of Reporting Problems

Quick Problem Solutions

If you have a problem with Lahey software and have access to a modem and our BBS, first download the ".FIX" file to test if we have already corrected the problem. Lists of resolved problems for all currently-supported product versions are maintained on the BBS. Versions of ".FIX" files for all currently-supported releases of Lahey software are maintained on the BBS. Currently-supported releases include the most recent Lahey release and any prior releases up to 60 days following the most recent release dates.

If the problem was not fixed by downloading the ".FIX" file, or you do not have access to a modem, please use one of the following procedures:

- **Telephone**
- **Mail and Common Carrier**
- **FAX and TELEX**

Telephone Procedure

1. Dial (702) 831-2500 between 9:30 AM and 3:30 PM Pacific Time to receive technical support. Remember, calls will not be received outside of these hours and support calls cannot be taken on the 800 sales and orders telephone line.

2. Tell the Lahey Technical Support Representative what product you are using and the serial and version numbers. Also tell them the type of hardware you are using, how it is configured and what operating system and version you are using.

3. Please explain the problem in as much detail as possible since we log all support calls.

4. If the problem cannot be rectified over the phone, our Technical Support Representative will ask you to send us an example of your problem. When sending us a problem report, please include the following:

   A. a description of the problem to help us duplicate it;

   B. a sample of the code that causes your problem. Please make the example as small as possible to shorten our response time and reduce the chance of any misunderstandings;

   C. a copy of the ".FIG" file (configuration file);

   D. the size and date of the ".FIX" file; and

   E. your name, address, telephone, Fax or Telex number and the Lahey product name, serial and version numbers.

5. See the instructions for sending your report via mail, Fax or Telex. Lahey will contact you with a post card when we receive your problem. Please do not call more than once a week unless you have something new to report that could expedite the process. Calling frequently ties up the phone lines for other users and may actually slow down any progress on fixing the problem. You will be notified when your problem is resolved.

Electronic Bulletin Board System Procedure

The BBS is available 22 hours a day on weekdays and 24 hours a day on weekends. It is "off line" for maintenance between 3:30 and 5:30 PM Pacific Time every weekday. To log on the BBS, use your modem and dial the BBS phone number. The BBS handles 300, 1200 and 2400 Baud-rate modems and automatically adjusts itself to the communication parameters, i.e, 8N1. Follow the BBS menu selections to upload and download files and messages. The first time you access the BBS, you are assigned a "guest" status which allows you to leave messages and download ".FIX" files for all currently-supported products. First time users should call Lahey or leave a BBS message requesting full access to the system. Once your product registration and serial number are confirmed, the BBS system supervisor will give you full access rights to the system.

1. Dial (702) 831-8023 and try the "Quick Solution" outlined above.

2. If the "Quick Solution" does not correct your problem, please include the following information when uploading your report.

   A. a description of the problem to help us duplicate it;

   B. a sample of the code that causes your problem. Please make the example as small as possible to shorten our response time and reduce the chance of any misunderstandings;

   C. a copy of the ".FIG" file (configuration file);

   D. the size and date of the ".FIX" file; and

E. your name, address, telephone, Fax or Telex number and the product name, serial and version numbers. Also include the type of hardware you are using, how it is configured and what operating system and version you are using.

3. Within two working days after leaving your message, Lahey will leave you a message on the BBS with either the solution to the problem or a schedule for providing the solution.

Mail and Common Carrier Procedures

1. If you are mailing us a problem report, please include the following:

   A. a description of the problem to help us duplicate it;

   B. a sample of the code that causes your problem on a disk. Please make the example as small as possible to shorten our response time and reduce the chance of any misunderstandings;

   C. a copy of the ".FIG" file (configuration file);

   D. the size and date of the ".FIX" file; and

   E. your name, address, telephone, Fax or Telex number and the product name, serial and version numbers. Also include the type of hardware you are using, how it is configured and what operating system and version you are using.

2. Mail your example disk and report to the following address:

Lahey Computer Systems, Inc.
Technical Support
P.O. Box 6091
Incline Village, NV 89450

3. Lahey will respond within one week after we receive your report with either the solution to the problem or a schedule for solving the problem.

FAX, TELEX and UUNET Procedures

1. The Lahey FAX (702) 831-8123, TELEX 9102401256 and UUNET (uunet!lahey!support or uunet!lahey!sales) lines are available 24 hours a day. When reporting a problem, please include the following in your transmission:

   A. a description of the problem to help us duplicate it;

   B. a sample of the code that causes your problem. Please make the example as small as possible to shorten our response time and reduce the chance of any misunderstandings;

   C. a copy of the ".FIG" file (configuration file);

   D. the size and date of the ".FIX" file; and

   E. your name, address, telephone, FAX, TELEX or UUNET number or address and the product name, serial and version numbers. Also include the type of hardware you are using, how it is configured and what operating system and version you are using.

2. Lahey will respond within one week after we receive your report with either the solution to the problem or a schedule for solving the problem.

I.5 Lahey Return Policy

Lahey Computer Systems, Inc. agrees to refund, to the registered user who purchased directly from Lahey, the entire purchase price of the product (including shipping charges), subject to the policies stated below. The refund for Lahey-authorized returns with "RGA" numbers will be completed within 15 days of Lahey receiving the returned materials. If you have purchased your Lahey Language System through a software dealer, then the return must be processed through that dealer.

Valid Reasons for Returning Lahey Software

- If a Lahey FORTRAN Language System is determined not to be a full implementation of the ANSI 77 Standard and Lahey does not fix the deviation from the standard within 30 days of the user's report.
- If Lahey software fails to perform with an approved, Lahey-compatible software package, and the problem has been created by the Lahey software, and Lahey does not fix the problem within 30 days of the user's report.
- If Lahey fails to fix a problem within 30 days of receiving the user's report.

Limitations on Refunds

- The user requesting the refund must be registered with Lahey as the owner.
- Lahey assumes no responsibility for any hardware or operating system problems not directly caused by the Lahey software.

Return Procedure

The user must report the reason for the refund request to a Lahey Computer Systems Technical Support Representative and receive a "Returned Goods Authorization" (RGA) number from Lahey. This "RGA" number must be clearly visible on the outside of the shipping carton.

The product being returned to Lahey must be postmarked within 15 days of receiving the RGA number from Lahey. The following files MUST BE DESTROYED by the user before returning the product for a refund:

- All copies of Lahey files delivered to you on the software disks.
- All object (".OBJ") files created with a Lahey Compiler.
- All executable (".EXE") files created by linking Lahey Compiler-generated object files.

- A signed statement so stating must be included with the returned software. Copy the format of the example below for this statement of compliance.

I, ______________________________(your name), in accordance with the terms specified in the Lahey Customer Relations Policy, acknowledge that I have destroyed all backup copies, executable files and object files created with the Lahey software. I no longer have in my possesion any copies of the returned files or documentation. Any violation of this agreement will bring legal action governed by the laws of the State of Nevada.

Signature ______________________________

Print Name ______________________________

Company Name ______________________________

Address ______________________________

____

Telephone ______________________________

Product ______________________________ Serial # __________

RGA Number ______________________________ Date ___/___/___

Refund check payable to ______________________________

<u>Shipping Instructions</u>

The user must package the software diskettes with the manual and write the "RGA" number on the outside of the shipping carton. Shipping charges incurred using US Mail Parcel Post will be reimbursed together with the purchase price. Overseas users will be reimbursed for return shipping charges up to a maximum of $10.00. Ship via US Mail Parcel Post to:

Lahey Computer Systems, Inc.
P.O. Box 6091
Incline Village, NV 89450-6091

1 OVERALL LANGUAGE CONSIDERATIONS

This chapter defines the notation used to present formally the F77L, F77L-EM/16 and F77L-EM/32 implementations of FORTRAN 77. Throughout this manual F77L, F77L-EM/16 and F77L-EM/32 will be referred to as F77L.

Also described are the syntax and elements of a compilable F77L source file.

1.1 Notation Used in This Manual

Words and phrases are underlined where they are defined.

Square brackets ([ ]) indicate optional items. The brackets are not part of the item and do not appear in correctly written FORTRAN statements.

An ellipsis (...) indicates that the preceding item may appear one or more times in succession. The ellipsis represents the position at which an item may be repeated.

The character ƀ represents a blank character and is used to clarify examples.

Example:

    'ƀƀABCƀ'

represents a sequence of six characters where the first two are blanks, the next three are "ABC" and the last character is a blank.

For illustration, throughout this manual UPPERCASE is used to represent FORTRAN Keywords, while lowercase is used for variable names or values that programmers supply.

Note the differences between Zero "0" and the letter "O".

If the general format of a statement is:

```
CALL name [([a[,a]...])]
```

the following forms are allowed:

```
CALL  name
CALL  name ()
CALL  name (a)
CALL  name (a,a)
```

1.1.1 Extensions to the FORTRAN Standard

Extensions to the ANSI X3.9 – 1978 FORTRAN Standard in this manual are printed in blue. This coloring is provided to help programmers differentiate between standard and non-standard FORTRAN statements and functions. Not all Lahey Language Systems support the same set of extensions. A complete list of extensions to the FORTRAN Standard supported by each product is provided in **Appendix N**.

1.1.2 CARRIAGE RETURN and ENTER Keys

When the instructions in this manual ask you to enter text, the required input is highlighted in **Boldface** type. You are required to press either the **RETURN** or **ENTER** key after the input to perform the desired operation.

1.2 Character Set

The F77L Character Set consists of uppercase letters, lowercase letters, digits and the following special characters:

'	**apostrophe**	(	**left parenthesis**
*	**asterisk**	–	**minus sign**
	blank	+	**plus sign**
:	**colon**	)	**right parenthesis**
.	**decimal point**	/	**slash**
=	**equal sign**		

The following characters are extensions to the standard:

&	**ampersand**	**<**	**less than**
$	**dollar sign**	**"**	**quotation mark**
!	**exclamation point**	**_**	**underscore**
>	**greater than**		

To improve readability, source statements may be written using blanks, uppercase and lowercase letters. Except within CHARACTER and Hollerith constants, the F77L Compiler ignores blanks and converts all letters to uppercase.

Appendix M presents the ASCII Character Set and collating sequence (e.g., A is less than Z, the blank character has the decimal value 32, "3" is less than "c", and "B" is less than "b").

All characters in the ASCII Character Set may be used in CHARACTER and Hollerith constants except control-Z (end-of-file) and control-M (carriage return). To enter these two characters in a CHARACTER constant, use concatenation and the CHAR function (see Sections 4.4 and 11.5).

1.3 Program Structure

A program unit, an ordered sequence of one or more FORTRAN statements, is either a main program or a subprogram. A main program controls the actions of an executable program; it may call subprograms to accomplish its goal. A subprogram is a program unit that can be used only when referenced by a main program or another subprogram; it cannot be executed by itself. **Chapter 10** describes the subprograms: subroutines, functions and block data subprograms.

A complete FORTRAN program is referred to as an executable program. An executable program is a collection of program units that consists of exactly one main program and zero or more subprograms. Program units may appear in any order in a source file and can be compiled separately.

The program unit that refers to a subprogram is known as the calling program. The calling program may be either a main program or another subprogram. When a subprogram is

invoked, it is said to be called by the calling program. Data items passed to and from a subprogram are known as arguments. Actual arguments refer to the values that are specified in the calling program and passed to a subprogram. Dummy arguments refer to the names that are specified in the subprogram to receive the passed values. Dummy arguments must have a one-to-one correspondence in type, number and order with the actual arguments passed to them.

1.3.1 Block Structures

A block structure is a DO loop or an IF...THEN...ELSE structure. To preserve the integrity of block structures, jumping into a block from outside the block is prohibited; and when blocks are nested, the inner block must be completely contained in the outer block. **Chapter 8** describes the block structures.

1.4 FORTRAN Statement Elements

A FORTRAN statement is a sequence of ASCII characters that define an optional statement label, keywords, labels, names, constants and operators. Statements may continue for more than one line.

NOTE: A statement may contain all blanks.

F77L accepts either standard-format (fixed field) source files or free-format source files. Both formats are defined later in this chapter.

1.4.1 Labels

A label is a nonblank sequence of at most five digits, one of which must be nonzero. Blanks and leading zeros are not significant in distinguishing between labels.

1.4.2 Statement Labels

Every FORTRAN statement may be identified with a unique label.

1.4.3 Keywords

Keywords indicate the beginning of FORTRAN statements and statement elements. Since keywords also meet the requirements for symbolic names, a particular sequence of characters is identified as a keyword or a symbolic name by the context in which it appears. There are no reserved words in FORTRAN. The keywords are:

ASSIGN	**END**	**LOGICAL**
BACKSPACE	**ENDFILE**	**OPEN**
BLOCK DATA	**END IF**	**PARAMETER**
CALL	**ENTRY**	**PAUSE**
CHARACTER	**EQUIVALENCE**	**PRINT**
CLOSE	**EXTERNAL**	**PROGRAM**
COMMON	**FORMAT**	**READ**
COMPLEX	**FUNCTION**	**REAL**
CONTINUE	**GO TO**	**RETURN**
DATA	**IF**	**REWIND**
DIMENSION	**IMPLICIT**	**SAVE**
DO	**INQUIRE**	**STOP**
DOUBLE PRECISION	**INTEGER**	**SUBROUTINE**
ELSE	**INTRINSIC**	**WRITE**

The following keywords are extensions to the standard:

BC EXTERNAL*	**HC EXTERNAL***	**MSC EXTERNAL***
CHAIN*	**INCLUDE**	**MS EXTERNAL***
DO WHILE	**INPUT**	**NAMELIST**
END DO	**LC EXTERNAL***	**OPTION BREAK**
		RECURSIVE

***NOTES:** The CHAIN statement is a feature supported in F77L and not in the extended-memory versions. Not all external statements are supported in all Lahey Language Systems. Refer to **Appendix I** and **Appendix K** for external statement information.

1.4.4 Constants

A constant is a value that does not vary. A constant's value and how it is to be manipulated are specified by the constant's syntax (see Section 2.2).

1.4.5 Names

A name is a letter optionally followed by a sequence of dollar signs, underscores and alphanumeric characters. Embedded blanks are ignored, and lowercase letters are converted to uppercase: e.g., the name "a b" is equivalent to the name "AB". Common block names may be from 1 to 28 nonblank characters. All other names may be up to 31 nonblank characters in length. Names identify constants, variables, arrays and global elements of a FORTRAN program.

Examples:

A	(same as a)
b_$	(same as B_$)
c$nAme$$	(same as C$NAME$$)

A constant name, declared in a PARAMETER statement, is a symbolic representation of a constant that may be referenced by using the name during the execution of the program unit in which it appears.

A variable is a name that may be assigned values of its type and referenced during execution of the program unit in which it appears.

An array is a name that identifies a contiguous, ordered set of data of a single type, the elements of which may be assigned values and referenced during execution of the program unit in which it is declared.

In FORTRAN, global names must be unique and must refer to either subprograms or common areas. Global names are names that may be referenced in other program units.

1.4.6 Operators

In FORTRAN there are four sets of operators that are used to construct numeric, relational, CHARACTER and LOGICAL expressions (see **Chapter 4**).

1.4.7 Expressions

Constants, names and operators are combined to form expressions. Expressions define numeric, relational, CHARACTER or LOGICAL operations that are to be performed. Expressions are described in **Chapter 4**.

1.5 FORTRAN Statements

A FORTRAN statement is either executable or nonexecutable.

Executable statements specify actions that are to be performed. The three classes of executable statements and the chapters where they are described are:

Statement	**Chapter**
assignment	7
control	8
input/output	9

Nonexecutable statements specify the arrangement of data in storage, the format of input/output data and the types of program units.

The nonexecutable statements and the chapters where they are described are:

Statement	**Chapter**
compiler-directive	5
specification	6
DATA	6
FORMAT	9
statement-function	10
PROGRAM and subprogram	10

1.5.1 Statement Order

Order of Statements and Comment Lines

<table>
<tr><td rowspan="5">Comment Lines

Blank lines

INCLUDE statement

OPTION BREAK statement</td><td colspan="3">PROGRAM, FUNCTION SUBROUTINE or BLOCK DATA statement</td></tr>
<tr><td rowspan="4">FORMAT and ENTRY statements</td><td rowspan="2">PARAMETER statements</td><td>IMPLICIT statement</td></tr>
<tr><td>other specification statements</td></tr>
<tr><td rowspan="2">DATA statement</td><td>Statement-function statements</td></tr>
<tr><td>Executable statements</td></tr>
<tr><td colspan="4">END statement</td></tr>
</table>

In a program unit, no statement may appear after a statement which is lower in the chart, e.g., a PARAMETER statement should not appear after a DATA statement within a program unit.

Comment lines may appear anywhere, including between lines of a continued statement

Blank lines may appear anywhere, including between lines of a continued statement.

The program units that make up an executable program may be contained in one source file or in several.

When source statements are in files separate from the main program, the INCLUDE statement (see Section 5.2) may be used to compile these source files into one object file.

1.6 Standard-Format Source

A standard-format source file is a sequence of variable-length records of ASCII characters, terminated by a carriage return followed by a line feed, in the format defined by the FORTRAN 77 Standard. The default extension of a standard-format source file is ".FOR". Standard-format source can also be specified on the command line with the "/NF" Compiler Option (see the **Compiler Options and Configuring** Section in **Appendix A**).

If a tab character appears in the first six columns, it is replaced by blanks through column 6 if the character following the tab is a letter, otherwise the tab is replaced by blanks through column 5 so that the character is placed in the continuation column (column 6). Tabs after column 6 are ignored, except in CHARACTER or Hollerith constants.

Example:

<tab>DATA array /'a','b','c',
<tab>1 'd','e','f'
<tab>2 ,'g','h'/

is equivalent to:

~~bbbbbb~~DATA array /'a','b','c',
~~bbbbb~~1 'd','e','f'
~~bbbbb~~2 ,'g','h'/

If a source line is less than 72 characters, it is extended with blanks to 72 characters (this is significant for CHARACTER and Hollerith constants). Any characters beyond column 72 are ignored.

Standard-Format Statement Labels

The statement label must be in columns 1 through 5 of the initial line of a statement.

Standard-Format Statement Continuation

A FORTRAN statement may be continued on one or more continuation lines. A continuation line is a non-comment line with a nonblank, nonzero character in column 6. The character in column 7 is processed as if it followed immediately after the character in column 72 of the previous line.

The number of continuation lines in a statement is limited only by the amount of dynamic storage available to the compiler. F77L will accept numbers of continuation lines far in excess of 19 (the number a FORTRAN 77 Compiler is required by the ANSI Standard to accept).

Standard-Format Comment Lines

A comment line is a line with a "c", "C", or "*" in column 1. Any characters may appear in any column after 1. Comment lines may appear anywhere within a FORTRAN program including between lines of a continued statement, but comment lines are not statements and may not be continued.

1.7 Free-Format Source

A free-format source file is a sequence of variable-length records of ASCII characters, terminated by a carriage return followed by a line feed, written without regard to any field or column boundaries. Text may begin anywhere within the source line. Source lines may be of any length. Tabs anywhere in a line, except in CHARACTER or Hollerith constants, are treated as blanks.

The default extension for free-format source files is ".F". Free-format source can also be specified on the command line with the "/F" compiler option (see the **Compiler Options and Configuring** Section in **Appendix A**).

Free-Format Statement Labels

A statement label, if present, is indicated when the first nonblank character on a source line is a digit. A statement label is terminated by the first nonblank, nondigit following it.

The following line has no statement label:

```
a = b*x + c
```

Here is the same line with the statement label 10:

```
10    a = b*x + c
```

Free-Format Statement Continuation

A FORTRAN statement may be written on a single line, or it may be continued onto one or more continuation lines. A continuation line is indicated by an ampersand (&) appearing as the first nonblank character.

A statement written with a continuation line is treated exactly as if the last character before the carriage return on the preceding line is followed immediately by the first character after the ampersand (&) on the continuation line. The carriage return/line feed and blanks to the left of the ampersand are discarded. This is important when CHARACTER and Hollerith constants are continued on multiple lines. In the following examples, <CR> represents a carriage return/line feed sequence.

The lines:

```
  total = 1 + 2 + 3<CR>
& + 4 + 5 + 6<CR>
&   +      7 + 8<CR>
```

are equivalent to:

```
total  = 1 + 2 + 3 + 4 + 5 + 6 +  7 +  8
```

and the lines:

```
  name = 'abc <CR>
&def<CR>
&gh'<CR>
```

are equivalent to:

```
name = 'abc  def gh'
```

Free-Format Comment Lines

A comment line is denoted by an asterisk (*) appearing as the first nonblank character. Comment lines may appear anywhere within a FORTRAN program. Comment lines are not statements and may not be continued; they may appear between the lines of a continued statement.

Examples:

```
1      i = 2
2      *This is a comment.
3         *  So is this.
4      J = 3
5 *    This is a comment.
6    & + 5
```

Line 6 is a continuation of the statement begun at line 4:

```
J = 3 + 5
```

NOTE: Free-format source file comment lines cannot begin with the letter "C" because of conflicts with statements like COMMON and CALL.

1.8 Trailing Comments

An exclamation point that is not part of either a comment or a CHARACTER or Hollerith constant indicates that the exclamation point and the text that follows to the end of the line is a trailing comment. A comment field including its "!" delimiter is processed as if it were a blank character. An exclamation point in column 6 in a standard-format FORTRAN source file indicates a continuation line except within a comment. Trailing comments should only be used after statements.

Examples:

```
  i = 2 ! This line has a comment field.<CR>
  name = 'abc<CR>
&def<CR>
&    gh'!So does this statement.<CR>
```

These lines are equivalent to

```
i = 2
name = 'abc  def gh'
```

1.9 Blank Lines and Blank Statements

A blank line is a line consisting of zero or more blanks.

Blank lines may even appear between continued lines.

A blank statement is a blank line with a statement label and may appear anywhere a statement may appear.

Example:

```
a)              i = 2
b)
c)  10
```

Line b is a blank line; and line c is a blank statement with the statement label 10.

NOTE: Other implementations of FORTRAN 77 may not allow blank lines between continuation lines or statement labels on blank lines.

RESERVED FOR YOUR NOTES

2 DATA TYPES and CONSTANTS

In FORTRAN, constants and the names of variables, arrays and functions are typed to determine the memory (bytes) their values require and the interpretation and operations on the values contained in that memory. This chapter describes each of the data types, the syntax for expressing constants of each type and the rule for the implicit typing of names. The type statements, which explicitly declare the types of named items, are described in Section 6.2.

2.1 Types

The syntax and semantics of the data type attribute described in this section are also applicable to descriptions of the IMPLICIT (Section 6.1), typed FUNCTION (Section 10.2) and type statements (Section 6.2).

The data types are:

CHARACTER	**INTEGER**
COMPLEX	**LOGICAL**
DOUBLE PRECISION	**REAL**

Associated with each type is an implicit length (determined by the ANSI FORTRAN Standard and the computer architecture) that specifies the number of memory bytes allocated for each entity of a given type and length.

F77L has additional lengths implemented to allow better hardware utilization.

The types, their implicit (conforming to the ANSI Standard) and optional lengths are listed in the following table.

Type	Implicit Length	Optional Length
INTEGER	4	2
REAL	4	note 1
DOUBLE PRECISION	8	note 1
COMPLEX	8	16
LOGICAL	4	1
CHARACTER	note 2	note 2

NOTES:

1. DOUBLE PRECISION and REAL*8 are synonyms. Each requires 8 bytes.

2. A CHARACTER datum may have length from 1 to 32768. If length is not explicitly specified, it is 1.

The type of a constant is determined by the syntax used to express that constant (see Section 2.2). The type of a variable, array or function is determined by the first appearance of the name of that entity in one of the following contexts:

1. explicitly in either a type statement (Section 6.2), or in a typed FUNCTION statement (Section 10.2); and

2. implicitly (Section 2.9), in a context where the type of the named entity is required: DATA, PARAMETER and any executable statement.

When a name appears in the following statements its data type is not determined: SAVE, COMMON, DIMENSION, ENTRY, EQUIVALENCE, EXTERNAL and non-typed FUNCTION. Other contexts that do not type a name are:

1. when the name is used as a FUNCTION statement argument; and

2. when the name is used as the implied-DO variable of a DATA statement.

2.2 Constants

The value of a constant does not vary. The syntax of the literal expressing a constant specifies both its value and type.

Example:

Constant	Type
1	INTEGER
1.0	REAL
1.0D01	DOUBLE PRECISION
(1.0,0.0)	COMPLEX
.TRUE.	LOGICAL
'1.0'	CHARACTER

Numeric constants are INTEGER, REAL, DOUBLE PRECISION or COMPLEX values.

An unsigned constant is a numeric constant without a leading sign. A signed constant is a numeric constant preceded by either a plus or a minus sign. An optionally signed constant is an INTEGER, REAL or DOUBLE PRECISION constant that is either signed or unsigned.

A hexadecimal constant is specified by the letter Z, followed by an apostrophe, followed by the constant which is a sequence of one to eight digits and the letters A, B, C, D, E and F and terminated by an apostrophe. Hexadecimal constants may only appear in value lists of DATA statements.

Examples:

```
Z'ab09'
Z'fedcba98'
```

A blank character occurring in a numeric or logical constant has no effect on the value of that constant.

Examples:

(1.0 , 1.0) and (1.0,1.0)

are equivalent.

' abc '   and   'abc'

are not equivalent because in a CHARACTER constant the blanks are significant.

2.3 INTEGER Type

An INTEGER (INTEGER*4) datum is an exact representation of an integer value. It may assume a positive, negative or zero value, but it must be an integral value within the range:

–2147483648 to 2147483647.

NOTE: The most negative integer, –2147483648, cannot appear as a constant in a source file, but it may be the result of calculations and input/output.

INTEGER*2 data have a range from –32768 to 32767.

The syntax of an INTEGER constant is an optional sign followed by one or more digits which are interpreted as a decimal number.

2.4 REAL Type

A REAL (REAL*4) datum is an approximation to the value of a REAL number which may be positive, negative or zero. A REAL datum has an absolute value range from 1.18E – 38 to 3.40E38 and a precision of one part in 10E7 (24 bits).

DOUBLE PRECISION (REAL*8) data have an absolute value range from 2.23E – 308 to 1.79E308 and a precision of one part in 10E16 (53 bits).

The syntax of a basic REAL constant is an optional sign, an integer part, a decimal point and a fractional part, in that order. Both the integer and fractional parts are sequences of digits; either, but not both, may be omitted.

Examples:

1.0
4.
.6
–100017.911
+.000000001

The syntax of a REAL exponent is the letter "E" (or "e") followed by an optionally signed integer constant. A REAL exponent denotes a power of ten.

Examples:

e01
e – 2
e + 2

The forms of a REAL constant are:

1. basic REAL constant;

2. basic REAL constant followed by a REAL exponent; and

3. INTEGER constant followed by a REAL exponent.

The value of a REAL constant that contains a REAL exponent is the product of the constant that precedes the "E" and the power of ten indicated by the integer following the "E".

Examples:

2.0e2	has the value 200;
.001e + 3	has the value 1;
.003e – 10	has the value .0000000000003;
–40e2	has the value –4000.

2.5 DOUBLE PRECISION Type

A DOUBLE PRECISION datum is an approximation to the value of a real number, but the accuracy of the approximation is greater than that of type REAL. A DOUBLE PRECISION datum may assume a positive, negative or zero value. It has an absolute value range from 2.23D – 308 to 1.79D308 and a precision of one part in 10D16 (53 bits).

The syntax of a DOUBLE PRECISION exponent is the letter "D" (or "d") followed by an optionally signed INTEGER constant. A DOUBLE PRECISION exponent denotes a power of ten. The form and interpretation of a DOUBLE PRECISION exponent are identical to those of a REAL exponent, except that the letter "D" is used instead of the letter "E".

Examples:

```
d 01
d + 03
d – 3
```

The forms of a DOUBLE PRECISION constant are:

1. basic REAL constant followed by a DOUBLE PRECISION exponent; and

2. INTEGER constant followed by a DOUBLE PRECISION exponent.

Examples:

```
12.00000000000001d + 2
1.0 d + 1
+.10001110001110234d – 32
3141592653589793d – 15
–47d0
```

2.6 COMPLEX Type

A COMPLEX datum is an approximation to the value of a COMPLEX number. The representation of a COMPLEX datum is in the form of an ordered pair of REAL data. The first of the pair represents the real part of the COMPLEX datum; the second represents the imaginary part. Each part has the same degree of approximation and the same range as a REAL datum.

The syntax of a COMPLEX constant is a left parenthesis followed by an ordered pair of optionally signed REAL or INTEGER constants separated by a comma and followed by a right parenthesis.

Examples:

```
(0,0)
(0, -1.)
(+2,314e - 2)
```

A COMPLEX*16 constant is a pair of DOUBLE PRECISION constants.

Example:

```
( 1.4d0, 0.1d0 )
```

2.7 LOGICAL Type

A LOGICAL datum LOGICAL*1 or LOGICAL*4 has either the value true or the value false. The syntax and values of a LOGICAL constant are:

Form	Value
.TRUE.	true
.FALSE.	false

2.8 CHARACTER Type

A CHARACTER datum is a non-empty, fixed-length ordered sequence of characters. The sequence may consist of any characters in the ASCII Character Set found in **Appendix M**.

The syntax of a CHARACTER constant is a sequence of one or more ASCII characters enclosed in apostrophes. These delimiting apostrophes are not part of the datum represented by the constant. An apostrophe within a CHARACTER constant delimited by apostrophes must be represented by two consecutive apostrophes with no intervening blanks.

In a CHARACTER constant, blanks embedded between the delimiters are significant. The length of a CHARACTER constant is the number of characters between the delimiters, except that each pair of the delimiters within a sequence counts as a single character.

Each character in the datum has a position numbered consecutively from the left, beginning with one (1, 2, 3, etc.). The number indicates the position of a character in the sequence.

F77L allows use of quotation marks as delimiters in addition to apostrophes. A quotation mark within a CHARACTER constant delimited by quotation marks is represented by two consecutive quotation marks with no intervening blanks.

CHARACTER constant examples:

Constant	**Represents**	**Length**
'abc'	abc	3
"abc"	abc	3
"a'bc"	a'bc	4
'a"bc'	a"bc	4
'a''bc'	a'bc	4
'a bc'	aƀƀbc	5

2.9 Implicit Typing

If a name that identifies a variable, array, external function or statement function does not appear in an explicit type statement (see Section 6.2), then it is typed implicitly. If the first letter of the variable name is an I, J, K, L, M or N, it is typed INTEGER. If the first letter of the variable name is any other letter, it is typed REAL. This implicit typing convention may be altered through the use of the IMPLICIT statement (Section 6.1).

Examples:

INTEGER	REAL
i	alpha
integer	bet
new	z

RESERVED FOR YOUR NOTES

3 ARRAYS

An array is a contiguous, ordered set of data, all of the same type, identified within a program unit by a single name. Each member of the contiguous ordered set is called an array element. The number and ordering of elements must be declared in every program unit in which the array appears.

3.1 DIMENSION Statement

A DIMENSION statement specifies the symbolic name and dimensions of one or more arrays. The statement syntax is:

```
DIMENSION ad[,ad]...
```

Where:

"ad" is an array declarator having the syntax:

```
name([d1:]d2 [,[d1:]d2...])
```

"name" is the symbolic name of the array;

"[d1:]d2" is a dimension specification;

"d1" is a lower dimension bound; and

"d2" is an upper dimension bound.

Each name appearing in a DIMENSION statement declares an array in that program unit. Array declarators may also appear in COMMON and type statements. A name may appear only once in all array declarators of a program unit.

Each pair of dimension bounds, "d1" and "d2", specifies information about one dimension of the array.

The lower and upper dimension bounds must be INTEGER expressions of constants, symbolic names of constants and variables (not array elements or functions). The value of the dimension expression may be negative, zero or positive. The value of the upper dimension bound must be greater than or equal to the value of the corresponding lower dimension bound. If only the upper dimension bound is specified, the value of the lower dimension bound is one. In the array:

```
list(100)
```

the array "list" has one dimension with subscripts from 1 to 100.

The array:

```
table(–1:10,–5:–1,4)
```

has three dimensions. The first has subscript values from –1 to 10, the second –5 to –1, the third from 1 to 4. The element "table(5,–2,3)" is in the array. The element "table(6,2,4)" is outside of the dimensions specified in the array declarator.

The last upper bound of a dummy array declarator (in a called subprogram) may be specified as *. In this case, the array is known as an assumed-size array, and there is no optional check to ensure the array element refers to a declared element. Such arrays may not be used in statements that reference the whole array in a nonargument context (e.g., in an input/output list). See **Referencing the Array as a Unit** in Section 3.3.

3.2 Properties of an Array

As described above, an array declarator specifies the following properties of an array: the number of dimensions of the array, the size and bounds of each dimension, and therefore, the number of array elements. The data type of the array must also be specified, implicitly or explicitly.

Data Type of an Array

The data type of an array (see **Chapter 2**) is specified in the same manner as the data type of a variable. If the array contains INTEGER or REAL values, or if an IMPLICIT statement specifies implicit typing for other data types, then the data type may be implicitly defined by the first letter of the array name. Otherwise, the array name must appear in a type statement.

Example:

```
DIMENSION table(1:10),i(-5:0)
DOUBLE PRECISION table
```

In these two statements, "table" is typed as DOUBLE PRECISION, and "i" is not typed.

Dimensions of an Array

The number of dimensions of an array is equal to the number of dimension declarators in the array declarator. The size of a dimension is the value:

d2 – d1 + 1

Where:

"d1" is the value of the lower dimension bound; and

"d2" is the value of the upper dimension bound.

NOTE: If the value of the lower dimension bound is one, the size of the dimension is "d2" (the upper dimension bound).

table(10)	has a dimension size of 10
list(–5:1)	has a dimension size of 7
list(0:0)	has a dimension size of 1

Number of Array Elements

The number of elements in an array is equal to the product of the sizes of the dimensions specified by the array declarator for that array name. The array:

list(5,10:20)

has 5 * (20 – 10 + 1) or 55 array elements.

Array Element Storage

Knowing the order in which array elements are stored is required for:

1. input or output of entire arrays;
2. for specification of initial values in DATA and type statements;
3. for control of storage allocation using EQUIVALENCE and COMMON statements; and
4. for passing subsets of array elements to subprograms.

A one-dimensional array is stored as a linear list. For arrays of higher dimensions, the rule is that the first subscript varies most rapidly; the second, next most rapidly; and so on, with the last varying least rapidly.

The array declarator:

c(2,0:1,–5:–4)

represents array storage

c(1,0, –5)
c(2,0, –5)
c(1,1, –5)
c(2,1, –5)
c(1,0, –4)
c(2,0, –4)
c(1,1, –4)
c(2,1, –4)

3.3 Array Subscripting

To reference an element of an array, use the array name and specify the element of the array. The information that designates the element is known as a subscript.

Referencing an Array Element

The form of an array element name is:

name(s[,s]...)

Where:

"name" is the array name;

"(s[,s]...)" is a subscript; and

"s" is a subscript expression.

The number of subscript expressions must equal the number of dimensions in the array declarator for the array name.

NOTE: The term "subscript" includes the parentheses that delimit the list of subscript expressions.

The following program:

```
        DIMENSION small tbl (6,2)
        DATA small tbl/6*2*0./
        small tbl (3,2) = 2.
        PRINT 5, small tbl
  5     FORMAT (6f7.2)
        END
```

prints

```
  0.00   0.00   0.00   0.00   0.00   0.00
  0.00   0.00   2.00   0.00   0.00   0.00
```

A subscript expression is a numeric expression that may contain array element references and function references. If the subscript expression is not type INTEGER, the value is converted to type INTEGER according to the **Arithmetic Conversion Rules** (see Section 7.1) before the value of the subscript is computed.

In the evaluation of subscript expressions, a function must not alter the value of any other subscript expression within the same statement (see Section 4.7).

The FORTRAN compiler checks that the number of arithmetic expressions in a subscript is the same as the number of dimensions declared for the array. The value of the subscript determines which array element is identified by the array element name. Each subscript expression value must be within the corresponding array bound declaration so that the element referenced by a subscript must be within the array.

The following program:

```
        REAL table (–4:1,2)
        DATA table/6*2*0./
        a = 3.0
        b = 2.0
        table (–(a*b/2),a – b) = 2.
        PRINT 6, table
  6     FORMAT (6f7.2)
        END
```

prints

0.00	2.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00

Referencing the Array as a Unit

Whenever an array name is not qualified by a subscript, its use designates the whole array. The appearance of the array name implies that the number of values to be processed is equal to the number of elements in the array and that the elements of the array are taken in sequential order. There may be a reference to an entire array (without subscripts) only in the following contexts:

1. in the list of dummy arguments in a FUNCTION, ENTRY or SUBROUTINE statement;
2. in the list of actual arguments in a reference to a subroutine or function;
3. in a type statement;
4. in a COMMON statement;
5. in an EQUIVALENCE statement;
6. in a SAVE statement;
7. in a DATA statement;
8. as an internal file unit identifier in a READ or WRITE statement;
9. as the format identifier in an input/output statement; and
10. in the list of an input/output statement.

An assumed-size array cannot appear in contexts 4 through 10.

The program:

```
        INTEGER array1 (3,2)
        DATA array1 /1,2,3,4,5,6/
        INTEGER array2 (3,2)
        EQUIVALENCE (array1,array2)
        CALL subr (array2)
        END

        SUBROUTINE subr (array2)
        INTEGER array2 (3,2)
        PRINT 11, array2
   11   FORMAT(' '3G8.2)
        END
```

prints

```
   1.00    2.00    3.00
   4.00    5.00    6.00
```

3.4 Adjustable Array Dimensions

An array that is passed to a function or subroutine may use INTEGER variables as dimensions. The variables can be passed to that subprogram as arguments or in a COMMON, in dimension-bound expressions. Dimensions so defined are known as adjustable dimensions. An array with adjustable dimensions refers to an array that has fixed, nonadjustable bounds in some calling program unit.

The execution of different references to a subprogram or different executions of the same reference determines possible different properties (size of dimensions, dimension bounds, number of elements and array element ordering) for each adjustable dimensioned array in the subprogram. These properties depend on the values of any arguments and variables in COMMON that are referenced in the adjustable dimension expressions in the subprogram.

The program:

```
              REAL table (100)
              INTEGER dimens /5/
              CALL sbrtn (table,dimens)
              END

              SUBROUTINE sbrtn (table,dimens)
              INTEGER dimens
              REAL table (dimens)
              DO 250 k = 1,dimens
              table (k) = k
250           CONTINUE
              PRINT *, table
              END
```

prints

```
1.000000    2.000000    3.000000    4.000000    5.000000
```

The array "table" had to be declared with fixed bounds in the main program before it could be used with adjustable dimensions in the subroutine.

NOTE: In a subprogram, FORTRAN allows an array that was passed as an argument to be dimensioned to be larger than the actual size of the array. F77L does subscript checking based on the size of the array as it was dimensioned in the subprogram. This means that one could inadvertently alter the area of memory which lies outside the true allocation of the array.

RESERVED FOR YOUR NOTES

4 EXPRESSIONS

An expression is a constant, variable, array element, substring, function reference or any combination of these joined by operators. The types of expressions in FORTRAN are:

> **numeric;**
> **character;**
> **relational;** and
> **logical.**

This chapter describes each of these types of expressions.

4.1 Operator Precedence

Operator precedence refers to the order in which an expression is evaluated. Unless changed by the use of parentheses, the order in which an expression is evaluated is determined by an established precedence of operators. For example, the operator "*" has a higher precedence than the operator "+". In all expressions that contain both the "*" and the "+" operators, the operand pairs associated with the "*" operator are evaluated first.

The precedence of all operators from the highest to the lowest is shown in the following table. With the exception of the relational operators, which all have the same precedence, all operators shown on the same line are at the same level of precedence.

Expression Type	Operators
numeric	**
	* , /
	+ , −
character	//
relational	.LT., .GT., .LE., .GE., .EQ., .NE.
logical	.NOT.
	.AND.
	.OR.
	.EQV., .NEQV.

NOTE: In F77L ".GT." and ">" and ".LT." and "<" are interchangeable.

In expressions with two or more operators of equal precedence, the expression is evaluated from left to right. Exponentiation, which is evaluated from right to left, is the exception.

The expression:

```
–2 + 6**2+4/2 + 3
```

is evaluated as

```
–(2) + (6**2) + (4/2) + 3
```

and

```
2**3**2
```

is evaluated as

```
2**(3**2)
```

Within an expression the results are determined from left to right, while functions and exponents are evaluated from right to left. In the expression:

```
f1(x)*f2(y)
```

the function "f2" is evaluated before "f1".

In a numeric expression, all exponentiation operations are evaluated first, then all multiplication and division operations, and finally all addition and subtraction operations. It is possible, however, to control the order of evaluation through the use of parentheses.

Examples:

Expression	Interpretation or Evaluation
a + b*c + d	(a + (b*c)) + d
–2 + 4 – 6**8/9	((–2) + 4) – ((6**8)/9)
3.14*5**6/(2 + 3 + 4)	(3.14*(5**6))/(2 + 3 + 4)
2**3**4	2**(3**4)
–a + b*c**2	(b*(c**2)) – a
+a + b – c + d – e	(((a + b) – c) + d) – e
a**b**c*d	(a**(b**c))*d
a*b/d*c	((a*b)/d)*c
a**b + c*d	(a**b) + (c*d)

Only the first operand in a numeric expression or an expression enclosed in parentheses may be preceded with the negation or identity operator. This precludes two or more consecutive numeric operators between operands, such as 2 + – 4. By using parentheses for grouping, one can construct numeric expressions such as a**(–b) and a + (–b).

4.2 Constant Expressions

A constant expression is either a constant or an expression whose operands are constants. Any exponents in a constant expression are of type INTEGER.

Examples:

```
2.0*4.D0 + 2
2.0*(4.0D + 2/6.0D + 3)
4.0
2 .GT. 3
```

An INTEGER constant expression is a constant expression in which all constants and symbolic names of constants are of type INTEGER:

```
3
–3
–3 + 4
–3*4**2
```

4.3 Numeric Expressions

A numeric expression specifies a numeric computation, and the evaluation of a numeric expression produces a numeric value (i.e., a value of type INTEGER, REAL, DOUBLE PRECISION or COMPLEX).

The numeric operands are constants, variables, array elements and function references.

Numeric Operators

The numeric operators are as follows:

Operator	Use	Interpretation
**	a1**a2	raise a1 to the power a2
/	a1/a2	divide a1 by a2
*	a1*a2	multiply a1 by a2
–	a1 – a2	subtract a2 from a1
–	–a1	negate a1
+	a1 + a2	add a2 to a1
+	+a1	same as "a1"

Each of the operators "**", "/" and "*" performs a binary operation on a pair of operands and is written between the two operands. The operators "+" and "–" are used in two ways:

1. as a binary operator on a pair of operands; or

2. as a unary operator on a single operand written preceding that operand, as in:

negation: –7

or

identity: +7

Types and Values of Numeric Expressions

The type of a numeric expression is determined from the types of the operands. When the negation or identity operator is used on a single operand, the data type of the resulting expression is the same as the data type of the operand. Examples:

–2	is of type INTEGER
+2.0	is of type REAL
–(2.0, –32.0)	is of type COMPLEX

The data type of the result of a numeric operator on a pair of operands is the higher type of the two operands in the following list:

COMPLEX*16
COMPLEX[*8]
REAL*8 or DOUBLE PRECISION
REAL[*4]
INTEGER[*4]
INTEGER*2

Example:

2 + 4.0

has a result of type REAL. The "2" is converted to type REAL before the addition, as if the expression had been "2.0 + 4.0". The expression:

2 + (4.0,6.0)

has a result of type COMPLEX.

When an expression is evaluated, operators of the same precedence are evaluated in a left to right order.

Example:

```
a + b*c/d
```

is evaluated as

```
a + ((b*c)/d)
```

As defined above, the type of each binary operation is the higher type of each of the two operands; this also is decided in a left to right order.

Example:

```
1 + 2.*i/j
```

is evaluated as

```
1 + ((2.*i)/j)
```

so that the expression is of type REAL; whereas

```
1 + 2*i/j
```

is evaluated as an INTEGER expression. In the expression:

```
3.0D02*(2 + 4.0)
```

the "2" is first converted to "2.0" (a REAL value) and then added to "4.0". The result, "6.0", is then converted to DOUBLE PRECISION and multiplied by "3.0D02".

Evaluation of Numeric Expressions

Any numeric operation whose result is not mathematically defined is prohibited. Thus, it is prohibited to:

1. divide by zero; or

2. raise a zero-valued operand to a zero-valued or negative-valued power; or

3. raise a negative-valued operand base to a REAL or DOUBLE PRECISION operand power.

Two numeric expressions are mathematically equivalent if, for all possible values of their operands, their mathematical values are equal. However, mathematically equivalent numeric expressions may produce different computational results. This is due to the finite manner in which the computer stores numeric data and to the round-off errors that may occur in evaluating two expressions.

The following examples show types of FORTRAN numeric expressions that may be considered equivalent and those that are mathematically equivalent but not computationally equivalent.

Expression	Equivalent Form
x + y	y + x
x*y	y*x
–x + y	y – x
x + y + z	x+ (y + z) or (x + y) + z
x – y + z	x – (y – z)
x*b/z	x*(b/z)
a/b/c	a/(b*c)
a/2.0	0.5*a

Expression	Nonequivalant Form
i/2	0.5*i
x*i/j	x*(i/j)
i/j/a	i/(j*a)
x*(y – z)	x*y – x*z

Parentheses may be included in an expression to establish the desired interpretation. This is useful for controlling the magnitude and accuracy of intermediate values developed during the evaluation of an expression. In the expression:

(a + (b – c))

the term "(b – c)" is evaluated first and then added to "a".

Integer Division

In FORTRAN the quotient of two integers is called an integer quotient and any remainder is truncated. Therefore, if the resulting magnitude of the integer division is less than one, the integer quotient is zero.

Expression	Value
2/4	0
4/1	4
(–8)/3	–2
99/100	0
–100/99	–1

4.4 CHARACTER Expressions

A CHARACTER expression defines a character sequence and produces a result of the CHARACTER type.

CHARACTER Operator

The only CHARACTER operator is concatenation, //. The form of an expression is:

x1//x2

Where:

"x1" denotes the operand to the left of the operator; and

"x2" denotes the operand to the right of the operator.

The result of a concatenation operation is a sequence of characters whose value is the value of "x1" on the left concatenated with the value "x2" on the right. The length of "x1//x2" is the sum of the lengths of "x1" and "x2".

Example:

    'ab'//'cde'

is the value "abcde" whose length is five.

NOTE: The concatenation operator is not commutative, i.e., 'a'//'b' is not equal to 'b'//'a'. All blanks are preserved, e.g., 'a~~b~~b~~b~~'//'~~b~~cde' is equal to 'a~~b~~b~~b~~~~b~~cde'.

<u>CHARACTER Operator Precedence</u>

Because concatenation is the only CHARACTER operator, there are no precedence rules for a CHARACTER expression.

Example:

    a//(b//c)

is identical to

    a//b//c

<u>CHARACTER Operands</u>

The <u>CHARACTER operands</u> are:

constant;
symbolic name of constant;
variable reference;
array element reference;
function reference (see **Chapter 11** for Intrinsic Functions);
substring reference; and
expression enclosed in parentheses.

Substrings

A substring is a variable or array element.

A substring is also a contiguous portion of a CHARACTER constant, a function result or a parenthetical expression.

A substring is identified by a substring name.

Substring Names

A substring name is local to a program unit. The forms of a substring name are:

```
v ([e1] : [e2])
a(s [,s]...) ([e1] : [e2])
f([a][,a]...) ([e1] : [e2])
c ([e1] : [e2])
p ([e1] : [e2])
```

Where:

> "v" is a variable name;
>
> "a(s[,s]...)" is an array element name; (see Section 3.3);
>
> "f([a][,a]...)" is a function reference;
>
> "c" is a constant;
>
> "p" is a parenthetical expression; and
>
> "e1" and "e2" are numeric expressions and are called substring expressions.

The integer value of "e1" specifies the left most character position of the substring. The integer value of "e2" specifies the rightmost character position of the substring.

Example:

```
v(2:4)
```

specifies a substring that contains the characters in positions 2 through 4 of the value of the variable "v". If "v" contains "abcdef", then "v(2:4)" contains "bcd".

Example:

```
b(1,2)(2:6)
```

specifies a substring that contains the characters in positions 2 through 6 of the character array element "b(1,2)". If:

```
b(1,2) = 'aaabbbccc'
```

then

```
b(1,2)(2:6) = 'aabbb'
```

The values of "e1" and "e2" must be such that "e1" is greater than zero; "e1" is less than or equal to "e2"; and "e2" is less than or equal to "len", where "len" is the length (number of characters) of the CHARACTER substring item. If "e1" is omitted, the value "1" is assumed. If "e2" is omitted, the value "len" is assumed.

Example:

Assuming the values:

```
a = 'abcdef'
b(1,2) = 'aaabbbccc'
len(a) = 6
len(b(1,2)) = 9
```

the following conditions are true regarding the substring notation:

a(0:2)	is illegal because "e1" is zero
a(1:7)	is illegal because 7 is greater than 6, the length of "a"

a(4:2)	is illegal because "e1" is greater than "e2"
a(:4)	is equivalent to "a(1:4)", which is "abcd"
b(1,2)(2:)	is equivalent to "aabbbccc", which is "b(1,2)(2:9)"
a(:)	is equivalent to "a(1:6)" and to "a"
b(1,2)(:)	is equivalent to "b(1,2)" and to "b(1,2)(1:9)"

Substring Expressions

A substring expression may be any numeric expression that may contain array element references and function references, but a function reference must not alter the value of any entities that appear in the same substring reference.

Example:

```
a(i*j:6)
```

is a substring expression where "i*j" is greater than or equal to 1 and less than or equal to 6. In the expression:

```
b(1,2) (2:i + j)
```

"i + j" must be greater than or equal to 2 and less than or equal to "len(b(1,2))".

Evaluation of CHARACTER Expressions

A CHARACTER expression is evaluated to determine a sequence of characters. A CHARACTER expression is a sequence of one or more CHARACTER operands separated by the concatenation operator (//). Parentheses have no effect upon the value of such an expression, but they may be used to improve readability. The following are equivalent:

```
'ab' // 'de' // 'ef'
('ab' // 'de') // 'ef'
'ab' // ('de' // 'ef')
'abdeef'(1:3) // 'abdeef'(4:6)
```

4.5 Relational Expressions

A relational expression is used to compare the values of two expressions. Evaluation of a relational expression produces a result of type LOGICAL. The value is ".TRUE.", if the values of the operands satisfy the relation specified by the operator. If the relation is not satisfied, the relational expression has the value ".FALSE.".

Relational Operators

The relational operators are the following:

Operator	Representing
.LT. or <	less than
.LE.	less than or equal to
.EQ.	equal to
.NE.	not equal to
.GT. or >	greater than
.GE.	greater than or equal to

No precedence exists among relational operators because a relational expression contains exactly one operator.

Example:

```
x .GE. 2.0
```

is legal, but:

```
x .GE. 2.0 .LE. y
```

is not.

Numeric Relational Expressions

The form of a numeric relational expression is:

e1 relop e2

Where:

"e1" and "e2" are numeric expressions; and

"relop" is a relational operator.

If "e1" or "e2" is of COMPLEX type, "relop" can only be .EQ. or .NE.

Be aware that since REAL and DOUBLE PRECISION values are approximations, relational expressions involving these values may not evaluate to the expected value.

CHARACTER Relational Expressions

The form of a CHARACTER relational expression is:

e1 relop e2

Where:

"e1" and "e2" are CHARACTER expressions; and

"relop" is a relational operator.

The expression "e1" is considered to be less than the expression "e2" if the value of "e1" precedes the value of "e2" in the collating sequence (see **Character Set**, Section 1.2). The value of "e1" is greater than the value of "e2" if the value of "e1" follows the value of "e2" in the collating sequence. For example, the collating sequence for the letters agree with their alphabetical order. See **Appendix M** for the complete collating sequence.

Example:

A	least
B	
C	
...	
Z	greatest

If "e1" = "CCC" and "e2" = "DDD", "e1" is less than "e2" because "C" precedes "D" in the collating sequence. If "e1" = "ABC" and "e2" = "ABB", "e1" is greater than "e2".

If the operands are of unequal length, the shorter operand is treated as if it were extended to the right with blanks, to equal the length of the longer operand.

4.6 LOGICAL Expressions

Evaluation of a LOGICAL expression produces a result of type LOGICAL and a value of either ".TRUE." or ".FALSE.". The basic forms of a LOGICAL expression are a LOGICAL constant, the symbolic name of a LOGICAL constant, a LOGICAL variable reference, a LOGICAL array element reference, a LOGICAL function reference or a relational expression. More complicated LOGICAL expressions may be formed by using one or more LOGICAL operands together with LOGICAL operators and parentheses.

LOGICAL Operators

The LOGICAL operators are:

Operator	Representing
.NOT.	negation
.AND.	conjunction
.OR.	disjunction
.EQV.	equivalence
.NEQV.	nonequivalence

The LOGICAL operator .NOT. is a unary operator; it has only one operand.

Examples:

```
.NOT. switch
.NOT. (L1 .AND. L2)
```

Truth Table for LOGICAL Operators

x1	x2	.NOT.x2	x1.AND.x2	x1.OR.x2
.TRUE.	.TRUE.	.FALSE.	.TRUE.	.TRUE.
.TRUE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.
.FALSE.	.TRUE.	.FALSE.	.FALSE.	.TRUE.
.FALSE.	.FALSE.	.TRUE.	.FALSE.	.FALSE.
x1	**x2**	**x1.EQV.x2**	**x1.NEQV.x2**	
.TRUE.	.TRUE.	.TRUE.	.FALSE.	
.TRUE.	.FALSE.	.FALSE.	.TRUE.	
.FALSE.	.TRUE.	.FALSE.	.TRUE.	
.FALSE.	.FALSE.	.TRUE.	.FALSE.	

The precedence of LOGICAL operators is:

Operator	Precedence
.NOT.	highest
.AND.	
.OR.	
.EQV. or .NEQV.	lowest

The precedence among the LOGICAL operators determines the order in which the operands are to be combined unless the order is determined by the use of parentheses:

Expression	Evaluation
a .OR. b .AND. c	a.OR.(b.AND.c)
a.OR. .NOT.b .AND.c	a.OR.((.NOT.b).AND.c)
a .AND. b .AND. c	(a .AND. b) .AND. c
a .OR. b .OR. c	(a .OR. b) .OR. c
a.OR.b .AND. c.OR.d	(a.OR.(b.AND.c)).OR.d
a.AND.b .OR. c.AND.d	(a.AND.b).OR.(c.AND.d)

LOGICAL Operands

The LOGICAL operands are:

constants;
variable references;
array element references;
function references;
relational expressions; and
expressions enclosed in parentheses.

Evaluation of LOGICAL Expressions

Two LOGICAL expressions are equivalent if their values are equal for all possible values of their operands. It is not necessary to evaluate all the operands of a LOGICAL expression if the value of the expression can be determined otherwise.

In evaluating the LOGICAL expression:

```
z .LE. w  .OR.  x .GT. y
```

the comparison "x" .GT. "y" is not executed to find the true value of the .OR. if "z" is less than or equal to "w".

4.7 Function References

The execution of a function reference (see Section 10.2) in a statement may not alter the value of any other entity within the statement in which the function reference appears. The execution of a function reference in a statement may not alter the value of any entity in common that affects the value of any other function reference in that statement. Results are undefined if either of these restrictions is violated. However, in a logical IF statement (Section 8.2) a function reference in the LOGICAL expression is permitted to affect the entities in the executable statement that follows the LOGICAL expression. If an argument of a function is defined as a result of that function evaluation, the argument may not appear elsewhere in the statement.

Example:

```
        REAL    a(5)
        DATA    x/100./, i/3/
  110   a(i) = f(i)
  120   y = 2 + x + g(x)
        ...
```

If the REAL function "f" assigns a value to its argument "i", then the statement labeled 110 is not defined in FORTRAN. Similarly, if the REAL function "g" sets "x", then the statement labeled 120 is not defined in FORTRAN.

COMPILER-DIRECTIVE STATEMENTS 5

SPECIFICATION and DATA STATEMENTS 6

ASSIGNMENT STATEMENTS 7

CONTROL STATEMENTS 8

STATEMENTS

5 COMPILER-DIRECTIVE STATEMENTS

Three FORTRAN statements which give directions to the compiler are described in this chapter. They are:

> **END;**
> **INCLUDE,** and
> **OPTION BREAK**

5.1 END Statement

Syntax

    END

Semantic Rules

The END statement indicates the end of the sequence of statements of a program unit. It must be the last statement in a program unit.

If the END statement is executed in a subprogram, it has the same effect as a RETURN statement (see Section 10.5). If the END statement is executed in a main program, it has the same effect as a STOP statement without an argument, i.e., it terminates the execution of the program (see Section 8.8).

5.2 INCLUDE Statement

Syntax

```
INCLUDE filename
```

Where:

"filename" is a character sequence, optionally enclosed by either quotation marks or apostrophes (as a CHARACTER constant), designating the name of a FORTRAN source file in the directory. If a filename extension is not specified, the compiler adds the extension ".F" if it is processing free-format source files, and ".FOR" otherwise. For more information on source filenames, see **Appendix A**.

Semantic Rules

The CHARACTER value designated by "filename" is the name of a file containing one or more FORTRAN source statements. The INCLUDE statement can be used to provide consistent declarations across a set of program units. An INCLUDE file may contain complete program units or fragments, but not incomplete statements.

The file designated by "filename" replaces the INCLUDE statement in the program being compiled. If the file "comfile.for" contains the following:

```
BLOCK DATA
COMMON/first/a,b,c
COMMON/second/d,e,f
DATA a,b,c,d,e,f/1,2,3,4,5,6/
END
```

then the following program:

```
CALL output
END
SUBROUTINE output
COMMON/first/a,b,c/second/d,e,f
PRINT *, a,f
END
INCLUDE comfile
```

prints

```
1.000000      6.00000
```

Nested INCLUDE Files

F77L permits one or more INCLUDE statements to appear in an included file. A file "bigfile.for" might be included in a program as follows:

```
PROGRAM main
INCLUDE bigfile
END
```

The file "bigfile.for" could contain the statements:

```
INCLUDE george
INCLUDE sam
INCLUDE harry
```

In nesting INCLUDE files care must be taken that the file references are not recursive. If, in the example above, "sam.for" contained:

```
INCLUDE bigfile
```

the compiler would loop endlessly between files "bigfile.for" and "sam.for", which INCLUDE each other.

5.3 OPTION BREAK Statement

OPTION BREAK allows the user to handle break interrupts during the execution of the program. A break interrupt occurs when either a control-C or control-break key combination is typed at the console. The system default action is the same as that of OPTION BREAK: all file buffers are flushed, and the program terminates with an error status.

If a break is received during console input/output, some data may be lost, and an error may occur on the input/output statement. Thus, if the user sets up a BREAK option which continues processing after a break (see OPTION NBREAK and OPTION BREAK (lvar) below), the console input/output error must also be trapped with an ERR= or IOSTAT= specifier on the input/output statement. Input/output on standard system files may be interrupted by breaks and continued without loss of data if OPTION NBREAK or OPTION BREAK (lvar) is used.

OPTION BREAK is executable, that is, it only takes effect when the statement is executed. Once executed, the BREAK Option remains in effect until another OPTION BREAK or NBREAK is executed in the same program unit or until the program unit containing the OPTION statement exits. It need not be executed after being invoked by a break interrupt in order to remain in effect. When a program unit returns, any BREAK Option executed in that unit is cancelled, and a subsequent break will be handled by any BREAK Option in effect in the calling unit or any higher program unit or else by the system default. Thus, a different BREAK Option may be set up in a subroutine, and it will supersede any BREAK Option executed in a higher program unit for the duration of that subroutine's execution.

The following variations on OPTION BREAK are provided:

OPTION BREAK

All file buffers will be flushed, and the program will terminate with an error status. This is the system default action.

OPTION NBREAK

Any break that occurs after this statement is executed will be ignored. Note, however, the previous discussion concerning interrupted console input/output.

OPTION BREAK (lvar)

Where:

> "lvar" is a LOGICAL variable which must have been defined in a common area or been named in a SAVE statement.

Breaks occurring after this statement do not interrupt the program, but "lvar" is set to ".TRUE." when a break is received. This allows selective testing of the variable and delayed handling of a break. Note again the previous discussion concerning interrupted console input/output.

OPTION BREAK (*label)

Where:

> "label" is a statement label in the same program unit as the OPTION statement.

If a break is received while this option is in effect, program control is transferred to the label. If file or console input/output is interrupted by the break, any data not yet transferred will be discarded, and no error will occur.

The following example shows a possible usage of BREAK options. When a break occurs, control is transferred to Statement 99 for the next command. In FILEWRIT, where file input/output takes place, BREAK (lvar) is used so that breaks are detected but do not interrupt the input/output.

```
          LOGICAL ERR*1
          OPTION BREAK (*99)
          READ *,COMMAND
          ...
          CALL FILEWRIT (ERR)
          ...
    99    PRINT *,'Command halted, next command'
          ...

          SUBROUTINE FILEWRIT (ERR)
          LOGICAL BREKRCVD, ERR*1
          SAVE BREKRCVD
          BREKRCVD = .FALSE.
          OPTION BREAK (BREKRCVD)
          ...
          ERR = BREKRCVD
          END
```

6 SPECIFICATION and DATA STATEMENTS

Specification and DATA statements are nonexecutable FORTRAN statements. Specification statements specify the type, value, reference method and allocation associated with a datum in a FORTRAN program.

The Specification Statements are:

COMMON	**INTRINSIC**
DIMENSION	**NAMELIST**
EQUIVALENCE	**PARAMETER**
EXTERNAL	**SAVE**
IMPLICIT	

The following specification statements are extensions to the FORTRAN 77 Standard. Not all Lahey products support all of these statements. Refer to **Appendix I** and **Appendix K** for further information on the supported statements for each product.

BC EXTERNAL	**LC EXTERNAL**	**MS EXTERNAL**
HC EXTERNAL	**MSC EXTERNAL**	

The following Specification Statements are Type Statements:

CHARACTER	**INTEGER**
COMPLEX	**LOGICAL**
DOUBLE PRECISION	**REAL**

The syntax and semantic rules for each of these statements, except DIMENSION, are described in this chapter. The DIMENSION statement is described in **Chapter 3**. The data types are described in **Chapter 2**.

DATA statements provide initial values for variables, arrays, array elements and substrings. The DATA statement is described in Section 6.10.

6.1 IMPLICIT Statement

Syntax

```
IMPLICIT NONE
IMPLICIT item [,item]...
```

Where:

"item" is of the form: type (a [,a]...)

"type" is one of:

INTEGER[*len], LOGICAL[*len],
REAL[*len], DOUBLE PRECISION,
COMPLEX[*len], CHARACTER[*len];

"len" is shown in the following table. In the table, the FORTRAN 77 Standard length is underlined; the standard length is also the implicit length.

Type	Length (Memory Bytes)
INTEGER	2 or <u>4</u>;
REAL	<u>4</u> or 8;
COMPLEX	<u>8</u> or 16;
LOGICAL	1 or <u>4</u>;
CHARACTER	<u>1</u>; see Section 6.2 for different ways to specify "len".

"a" is either a single letter or a range. A range is denoted by the first and last letters of the range, in alphabetical order, separated by a hyphen (–).

Semantic Rules

An IMPLICIT statement changes or confirms the default implied INTEGER and REAL typing (see Section 2.9). IMPLICIT statements specify a type for all otherwise untyped variables, arrays and functions (except intrinsic functions) whose names begin with any letter that appears in the specification. The specification has the form of either a single letter or a range of letters. For all but DOUBLE PRECISION data, an IMPLICIT statement may optionally specify lengths for data.

IMPLICIT statements do not change the types of intrinsic functions. An IMPLICIT statement applies only to the program unit that contains it. Type specification by an IMPLICIT statement may be overridden or confirmed for any variable, array or function name by the appearance of that name in a type statement. An explicit type specification in a FUNCTION statement overrides an IMPLICIT statement for the name of that function subprogram.

In a program unit, IMPLICIT statements must precede all other specification statements except PARAMETER or ENTRY statements (see **Chapter 10** for an explanation of ENTRY statements). Also, in a program unit, a letter must be assigned a type only once in the IMPLICIT statements.

The IMPLICIT NONE statement does not have an item list and must be the only IMPLICIT statement in the program unit. When NONE is specified, implicit typing is not permitted in the program unit; all variables, arrays and functions used (except intrinsic functions) must be explicitly defined by means of type statements.

If the following statements are given:

```
IMPLICIT REAL (m), REAL*8(d)
IMPLICIT INTEGER*2 (j, k), COMPLEX (i)
IMPLICIT CHARACTER*8 (f – g)
IMPLICIT CHARACTER (c)
INTEGER igor
```

then the names below would have the indicated types:

Variable	Type	Length
alpha	REAL	4
chart	CHARACTER	1
igor	INTEGER	4
k	INTEGER	2
im	COMPLEX	8
donald	REAL	8
f	CHARACTER	8

6.2 Type Statements

Syntax

A type statement has the form:

type v[*len][/i/][,v[*len][/i/]]...

Where:

"type" is one of:

INTEGER[*len], CHARACTER[*len],
REAL[*len], DOUBLE PRECISION,
COMPLEX[*len] LOGICAL[*len];

"v" is a variable name, array name, array declarator, function name or dummy procedure name;

"len" is an INTEGER datum; allowable values depend upon "type"; and

"i" is an initialization value.

Semantic Rules

A type statement is used to override or confirm implicit typing (see sections 2.9 and 6.1). A type statement may also specify dimension and initialization information. There is a type statement for each data type.

The appearance of the symbolic name of a variable, array or function in a type statement specifies the data type for that name each time that name appears in the program unit. Within a program unit, a name need not have its type explicitly specified more than once (redundant typing is ignored). Subroutine names, main program names and block data names must not appear in type statements.

If an intrinsic function name appears in a type statement that causes a conflict with the type specified by the definition of that function (see **Chapter 11**), the name loses its intrinsic function property in the program unit that contains that type statement. If the type statement:

```
INTEGER cos, sin
```

appears in a program unit, all uses of the names "cos" and "sin" would refer to INTEGER entities, not to intrinsic functions that determine the cosine and sine of an argument.

In a type statement, each item may be initialized by following the name or array declarator with an initial value (or values, for arrays) contained between slashes (Section 6.10). The statement:

```
INTEGER table(0:9)/0,1,2,3,4,5,6,7,8,9/
```

initializes the integer array "table" with the values 0 through 9. The statement:

```
CHARACTER bob*3/'ann'/,b,a/'a'/
```

initializes "bob" and "a", but not "b".

NOTE: Function names and dummy arguments may not be initialized.

Examples:

```
REAL x,w/1.0/
COMPLEX y/(1.0,1.0)/,C*16
LOGICAL z,flag*1/.FALSE./
INTEGER*2 a(4)/0,1,2,3/,i,j*4,k
```

Syntax

The form of a CHARACTER type statement is:

CHARACTER[*len[,]] name[,name]...

Where:

"name" is one of the forms:

v[*len][/i/]

or

a[(d)][*len][/i/]

"v" is a variable name, symbolic name of a constant, function name or dummy procedure name;

"a" is an array name;

"(d)" is a dimension declarator;

"len" is the length (number of characters) of "v" or each element of "a" and is called the length specification. It is one of the following:

1. an unsigned, nonzero, INTEGER constant;

2. an INTEGER constant expression, enclosed in parentheses and having a positive value;

3. an asterisk in parentheses, (*).

"i" is an initialization value (or values).

NOTE: If "v" is either a function or dummy name, initialization is not allowed.

Semantic Rules

A length of "*len" immediately following the word CHARACTER is the length specification for each entity in the statement not having its own length specification. If no length is specified following the keyword "CHARACTER", the length is implicitly 1. A length specification following an entity is the length specification for only that entity. If a length is not specified for an entity, its length is 1.

For an array, the length specified applies to each array element. In the statement:

```
CHARACTER*16, d, c*8, a(2)*4, b, e*9, f*9
```

"d" and "b" are of length 16; "c" is of length 8; "a(1)" and "a(2)" are of length 4; and "e" and "f" have lengths of 9.

If an entity declared in a CHARACTER statement has a length specification, the length specification must be an INTEGER constant expression, unless that entity is a parameter, dummy argument or a function.

If a dummy argument has a "len" of (*) declared, the dummy argument assumes the length of the associated actual argument for each reference of the subroutine or function. If a parameter has a "len" of (*) declared, the datum assumes the length of the constant CHARACTER expression assigned to the datum in the PARAMETER statement.

Adjustable Lengths

If a function has a declared "len" of (*), that function name must appear as the name of a function in a FUNCTION or ENTRY statement in the same program unit. When a reference to such a function is executed, the function assumes the length specified in the referencing program unit. The length specified for a CHARACTER function in a program unit referencing that function must be equal to the length specified in the CHARACTER function subprogram unit.

Dummy arguments are defined at entry time. A dummy argument length need not be defined in the same manner as a corresponding actual argument length, but the dummy argument length must be less than or equal to that of the actual argument.

NOTE: An array dummy argument has the length of the array, not just one array element. A specification of (*) defines the dummy character length to be that of a single array element.

Example:

```
CHARACTER * 10 actarg
CALL subroutn(actarg)
...
END

SUBROUTINE subroutn (dumarg)
CHARACTER dumarg (3)*3
```

or

```
CHARACTER dumarg *6
```

or

```
CHARACTER*10 dumarg
```

or

```
CHARACTER*(*) dumarg
```

show various ways to declare the lengths of dummy arguments corresponding to the one actual argument, "actarg". Alternately, the statement:

```
CHARACTER dumarg(4)*3
```

would be illegal since the length of the entire array (12 bytes) would be greater than the corresponding actual argument (10 bytes). There is always agreement of length if a "len" of (*) is specified.

6.3 PARAMETER Statement

Syntax

PARAMETER (pname = c[,pname = c]...)

Where:

"pname" is a symbolic name; and

"c" is a constant expression.

Semantic Rules

Each "pname" is defined to be the value of the constant "c" that appears to the right of the equal sign. Such a name is known as a parameter or the symbolic name of a constant.

Once each symbolic name is defined, that name may appear subsequently in that program unit as a constant in an expression or as a value in a DATA statement (see Section 6.10). The name of a constant may not appear as part of a FORMAT statement or format specification (see **Chapter 9**). A constant name may appear as part of another constant (see the example below).

A symbolic name in a PARAMETER statement identifies the corresponding constant only in that program unit. The data type of the symbolic constant depends on the type of the symbolic name and follows the same rules for data conversion that apply to assignment statements (see **Arithmetic Conversion Rules** in Section 7.1).

If the symbolic name identifies a CHARACTER constant, the constant expression "c" is left-justified and padded with blanks or truncated as necessary to produce a CHARACTER constant of the correct length.

The intrinsic function CHAR with a constant INTEGER argument is allowed in PARAMETER statement expressions.

Examples:

The following program:

```
IMPLICIT COMPLEX(c)
REAL i
PARAMETER(i = 1,c = i + 1)
PRINT *, i,c
END
```

prints

```
 1.000000     (2.000000, 0.000000)
```

The following program:

```
  CHARACTER * 5 long, short
  PARAMETER (long = '123456',
&short = '1234')
  PRINT *, long
  PRINT *, short // 'xxx'
  END
```

prints

```
12345
1234ƀxxx
```

The following program:

```
  LOGICAL true, false, maybe
  INTEGER small, large
  PARAMETER (small = 0, large = 1)
  PARAMETER (true = .TRUE.,
&false = .FALSE.)
  PARAMETER (maybe = (small .LT. large .AND.
&true .OR. false))
  PRINT *, true, false, maybe
  END
```

prints

```
T F T
```

The PARAMETER statement is often used to set dimensions and the initial, limit and incremental values of DO loops:

```
          PARAMETER (ndimen = 3)
          REAL a (ndimen)
          DO 10 i = 1,ndimen
10        a(i) = 0.0
```

6.4 COMMON Statement

Syntax

COMMON [/[cb]/]nlist[[,]/[cb]/nlist]...

Where:

"cb" is a common block name; and

"nlist" is a list of variable names, array names and array declarators.

Semantic Rules

Each omitted "cb" specifies the blank common block; if the first "cb" is omitted, the first two slashes are optional. In each COMMON statement, the entities occurring in an "nlist" following a block name "cb" (or omitted "cb") are declared to be in common block "cb" (or blank common). Names of dummy arguments must not appear in the list. A common block name "cb" may be referenced by any program unit in an executable program.

Any common block name "cb" (or an omitted "cb" for blank common) may occur more than once in one or more COMMON statements in the same program unit. The "nlist" following each successive specification of the same common block is treated as a continuation of the "nlist" for that common block. Entities in a common block (named or blank) are stored sequentially in order of their appearance in the "nlist" lists in that program unit, and in order of the cumulative appearance of such lists in COMMON statements of a program unit. The size of a blank common is the maximum size of the blank common over all the program

units in which it is declared. Named common blocks SHOULD be the same size in each program unit in which they are declared.

An item in named common storage may be assigned an initial value only in a DATA or type statement in a block data subprogram; items in blank common storage may not be assigned initial values.

Because association is by storage unit rather than by name, the names of the variables and arrays may be different in the program units that reference that common block. However, note that since types may be mixed in a common block, care must be taken to reference entities in a common block by names of the same data type. For example, if the first entity in blank common storage is of type REAL, then all program units should reference the first entity in blank common as a REAL variable. An exception to this is that either part of a COMPLEX (COMPLEX*16) datum may be referenced separately as a REAL (REAL*8) datum.

Example:

The following program:

```
INTEGER main1(3)
REAL main2
CHARACTER main3 *5
COMMON main1, main2, main3
main1(2) = 100
main2 = 14.5
main3 = 'abcde'
CALL subroutn
END

SUBROUTINE subroutn
INTEGER sub1(3)
REAL sub2
CHARACTER sub3 *5
COMMON sub1, sub2, sub3
PRINT *, sub1(2)
PRINT *, sub2
PRINT *, sub3
END
```

prints

```
100
14.50000
abcde
```

The COMMON statement in the main program unit (line 4) allocates the first three storage units of blank common to array "main1", the next unit to "main2" and the following two units to "main3". The subroutine then references the same storage locations with "sub1", "sub2" and "sub3".

COMMON statements are cumulative. The two statements:

```
COMMON a,b,c
COMMON d
```

are equivalent to

```
COMMON a,b,c,d
```

The following are equivalent:

```
DIMENSION a(10),b(2,6),c(2:10)
COMMON a,b,c
```

and

```
DIMENSION b(2,6),c(2:10)
COMMON a(10),b,c
```

and

```
COMMON a(10),b(2,6),c(2:10)
```

The following is an example of named and blank common storage usage:

```
PROGRAM main
COMMON i,j
COMMON/one/a,b,c/two/d,e,f,g
...

SUBROUTINE sub1
COMMON/one/x,y,z//k,l
...

SUBROUTINE sub2
COMMON m,n/two/o,p,q,r
...
```

Here "a", "b" and "c" share storage units with "x", "y" and "z". The variables "d", "e", "f" and "g" share storage units with "o", "p", "q" and "r". In blank common, "i", "k" and "m" share the same storage unit and "j", "l" and "n" share the same storage unit.

NOTE: The use of two slashes in subroutine "sub1" indicates blank common by the omission of a common block name.

6.5 EQUIVALENCE Statement

The EQUIVALENCE statement specifies the sharing of storage locations by two or more entities in a program unit. This allows the same storage unit to be referenced by different names within a program unit. It also saves storage by using the same area for first one value set, then another.

Syntax

```
EQUIVALENCE (nlist) [,(nlist)]...
```

Where:

"nlist" is a list of two or more variable names, array element names, array names or CHARACTER substring names separated by commas. Neither names of dummy arguments nor names of functions may appear in the list.

Example:

```
EQUIVALENCE (a,b,c),(d,e)
```

"a", "b" and "c" are associated and "d" and "e" are associated.

Semantic Rules

An EQUIVALENCE statement assigns two or more variables within the same program unit to the same storage location.

If the associated entities are of different data types, the EQUIVALENCE statement does not cause type conversion nor does it imply mathematical equivalence. If a variable and an array are associated, the variable does not have array properties and the array does not have the properties of a variable.

An EQUIVALENCE statement must not cause the storage sequence of two different common blocks in the same program unit to be associated. Equivalence association must NOT cause a common block storage sequence to be extended by adding storage units preceding the first storage unit of the first entity in that common block. The following is NOT permitted:

```
COMMON /s/a
REAL b(3)
EQUIVALENCE (a,b(2))
```

Every subscript and substring expression associated with a name in the list "nlist" must be an INTEGER constant expression.

An EQUIVALENCE statement specifies that the storage sequences of each of the entities in a list "nlist" occupy the same first storage unit. An EQUIVALENCE statement must NOT specify that the same storage unit is to occur more than once in a storage sequence.

The statements:

```
DIMENSION a(2)
EQUIVALENCE (a(1),b), (a(2),b)
```

are prohibited since they would specify the same storage unit for "a(1)" and "a(2)".

An EQUIVALENCE statement must not specify that consecutive storage units are to be nonconsecutive. The following is prohibited because it causes "d(1)" to overlap "d(2)".

```
REAL a(2)
DOUBLE PRECISION d(2)
EQUIVALENCE (a(1),d(1)),(a(2),d(2))
```

When an array element name appears in an EQUIVALENCE statement, the number of subscript expressions must equal the number of dimensions in the array declaration for that array.

The use of an array name that is unqualified by a subscript in an EQUIVALENCE statement is the same as using an array element name that identifies the first element of that array.

An entity of type CHARACTER may be "equivalenced" only with other entities of type CHARACTER; the lengths of the associated entities are not required to be the same. If CHARACTER items of different lengths are associated, the data are overlapped.

Example:

```
CHARACTER a*4, b*4, c(2)*3
EQUIVALENCE (a,c(1)), (b,c(2))
```

causes the following association of "a", "b" and "c":

	1	2	3	4	5	6	7
a	X	X	X	X			
b				X	X	X	X
c	X	X	X	X	X	X	

If the substring "b(1:1)" changes, so do the substrings "a(4:4)" and "c(2) (1:1)". If the value of "a" is "aaaa", the value of "b" is "abbb", the value of "c(1)" is "aaa", and the value of "c(2)" is "abb", then the CHARACTER assignment statement:

```
b(1:1) = 'f'
```

causes the value of "a" to be "aaaf", the value of "b" to be "fbbb", the value of "c(1)" to be "aaa", and the value of "c(2)" to be "fbb".

6.6 NAMELIST Statement

Use the NAMELIST statement to define a set of variable and array names and a unique namelist name by which the set will be known. The namelist name can then be used in namelist-directed input/output statements in lieu of format specifiers and input/output lists.

Syntax

```
NAMELIST/nlname/nllist[[,]/nlname/nllist]...
```

Where:

"nlname" is a namelist name; and

"nllist" is a namelist having the form:

```
va[,va]...
```

Where:

"va" is a variable or array name.

Semantic Rules

The NAMELIST statement is a specification statement and as such must precede the first executable statement of the program unit. A NAMELIST statement must also precede any statement function definitions.

The following rules apply to using the NAMELIST statement.

A namelist name defined in one program unit is independent of a namelist name defined in another program unit.

A namelist name must conform to the FORTRAN naming rules, (See **Names** in **Chapter 1**). As with subroutine names, no type is associated with a namelist name.

A namelist name must not be the same as a variable, array or procedure name in the same program unit. A namelist name is enclosed in slashes. A namelist of variable and array names is complete when either another namelist name enclosed in slashes or the end of the NAMELIST statement occurs.

A namelist name must be declared in one and only one NAMELIST statement of the program unit. After the declaration, the namelist name may appear only in input/output statements.

A variable or array name may belong to more than one namelist.

The order of variable and array names in a namelist determines the order in which values are written in namelist-directed output. The namelist-directed input of values can occur in any order.

A dummy array having a dimension of "*" may not appear in a namelist.

Array elements are not allowed in a namelist, but namelist-directed input can be used to assign values to elements of arrays that appear in namelists.

The rules for input/output conversion of namelist data are the same as the rules for data conversion shown in **Chapter 9** under **List-Directed Input and Output**.

A name that has been equivalenced to a variable or array name in a namelist cannot be used in lieu of the variable or array name in the list.

See Section 9.8 – **Namelist Input/Output**, for information on using the NAMELIST statement.

6.7 EXTERNAL Statement

Syntax

```
EXTERNAL proc [,proc]...
```

Where:

"proc" is the name of an external procedure, dummy procedure or block data subprogram.

Semantic Rules

If an external procedure name appears only as an actual argument in a program unit, it MUST also appear in an EXTERNAL statement in that program unit.

If a user-defined function or subroutine uses the same name as an intrinsic function, the appearance of that name in an EXTERNAL statement indicates that FORTRAN is to use the user-defined procedure, not the intrinsic function.

Example:

```
EXTERNAL sin
...
y = sin(x + 2.0)
...
```

"sin" refers to an external procedure written by the user and NOT to the FORTRAN intrinsic function (which could not be used in this program unit, but could be invoked by other program units in the executable program).

Because it is not an external procedure, a statement function name must not appear in an EXTERNAL statement.

To cause a block data program unit to be loaded from a library, the block data program unit name MUST appear in an EXTERNAL statement of a program unit that is loaded.

6.8 INTRINSIC Statement

Syntax

```
INTRINSIC fun [,fun]...
```

Where:

"fun" is an intrinsic function name.

Semantic Rules

A symbolic name must not appear in both an EXTERNAL and an INTRINSIC statement in the same program unit. The appearance of a name in an INTRINSIC statement declares the name to be an intrinsic function name that is to be used as an actual argument in the program unit.

Example:

```
INTRINSIC SIN
...
CALL JOE(SIN,5.0)
```

The names of the following intrinsic functions must NOT be used for actual arguments:

1. Type conversion (INT, INT2, INT4, IFIX, IDINT, FLOAT, SNGL, REAL, DBLE, CMPLX, ICHAR, CHAR);

2. Logical relationship (LGE, LGT, LLE, LLT);

3. Choosing the largest or smallest value (MAX, MAX0, AMAX1, DMAX1, AMAX0, MAX1, MIN, MIN0, AMIN1, DMIN1, AMIN0, MIN1); and

4. F77L character, bit manipulation and argument passing (NBLANK, CHARNB, NARGS), random number (RND, RRAND, RANDS), Boolean functions (IAND, IEOR, IOR, NOT), ISHFT, OFFSET, SEGMENT, POINTER, and CARG.

6.9 SAVE Statement

Local variables and arrays in subprogram units are not defined after execution of the subroutine or function. Many existing programs will not execute correctly because the subprogram expects values set in local items to be available the next time the subprogram is executed.

To solve this problem, the following alternatives are available:

Configure your compiler to REMEMBER local items by using "/R" in the F77L.FIG File;

Use "/R" on the command line; or

Add a SAVE statement for each affected subprogram unit.

Syntax

SAVE [a[,a]...]

Where:

"a" is the name of a named common block (preceded and followed by a slash), a variable or an array. A name must appear only once.

Example:

```
SAVE a,b,c
SAVE /com/
```

Semantic Rules

A SAVE statement without a list is treated as though it contained the names of all allowable noncommon items in that subprogram. Dummy argument names, procedure names and names of entities in a common block must NOT appear in a SAVE statement.

The appearance of a common block name in a SAVE statement has the effect of saving all entities in that common block.

The entities in a named common block may become undefined or redefined by executing a program unit in which the named common is specified. Entities in a SAVE statement retain their current values until the next time a subprogram in which they are declared is executed.

A SAVE statement is ignored in a main program since all common blocks, local arrays and variables of a main program are always saved.

6.10 DATA Statement

A DATA statement provides initial values for variables, arrays, array elements of all types and also for CHARACTER substrings. A DATA statement is nonexecutable and may appear anywhere in a program unit after the specification statements for the data list items.

Syntax

    DATA nlist/clist/[,nlist/clist/]...

Where:

"nlist" is a list of variable names, array names, array element names, substring names and implied-DO lists (see below);

"clist" is a list of the form:

    a[,a]...

"a" is one of the forms:

    c
    r * c

"c" is a constant or the symbolic name of a constant; and

"r" is the repeat count which is a nonzero unsigned INTEGER constant or the name of an INTEGER constant. Using "r*c" is equivalent to listing the constant, "c", "r" times in the "clist".

"r" is also the name of nonzero INTEGER constants.

Semantic Rules

DATA statements initialize the listed items only before the program begins execution. Dummy arguments, functions and entities in blank common (including entities equivalenced with an entity in blank common) must not appear in "nlist". Names of entities in a named common block may appear in the list "nlist" within a block data subprogram only.

There is a one-to-one correspondence between the items specified by "nlist" and the constants specified by "clist", such that the first item of "nlist" corresponds to the first constant of "clist", and so on. If an array name without a subscript is in the list "nlist", there must be one constant for each element of that array.

For CHARACTER or LOGICAL data types, the data type of the "nlist" entry must agree with the data type of its corresponding "clist" constant. However, an INTEGER, REAL, DOUBLE PRECISION or COMPLEX item in "nlist" can be initialized with an INTEGER, REAL, DOUBLE PRECISION or COMPLEX constant in the "clist"; the necessary conversion is performed automatically. Also, an INTEGER, REAL or DOUBLE PRECISION item (but NOT a COMPLEX item) may be initialized with a CHARACTER constant.

Example:

```
COMPLEX c
REAL d(4)
DOUBLE PRECISION e
DATA i,j/1,2/
DATA c,d,e/3,2.1,2.2,2.3,2.4,1.0/
```

"c" is initialized to (3,0); "d" to 2.1, 2.2, 2.3, 2.4; and "e" to 1.0.

A CHARACTER item in the list "nlist" can be initialized only with a CHARACTER constant. If the length of the CHARACTER entity in the "nlist" is greater than its corresponding constant in the list "clist", blanks are appended to the right of the constant to make its length equal to its corresponding CHARACTER entity. If the constant is longer, the surplus characters at the right of the constant are ignored. In this manner, initialization of a

CHARACTER entity causes initialization of every character in that entity. Each CHARACTER constant intializes exactly one variable, array element or substring.

It is possible to initialize a substring using the DATA statement:

```
CHARACTER*5,table
DATA table(3:5)/'ABC'/
```

Only the 3rd, 4th, and 5th character positions of the variable "table" are initialized. Character positions 1 and 2 are not defined.

Implied-DO in DATA Statements

An implied-DO list in a DATA statement may be used to initialize all or some of the elements of an array. The syntax is:

```
DATA nlist/clist/[,nlist/clist/]...
```

Where:

"nlist" can be an implied-DO list of the form:

```
(dlist, i = m1, m2 [,m3])
```

"dlist" is a list of array element names and/or implied-DO lists;

"i" is the name of an INTEGER variable, called the implied-DO variable; and

"m1", "m2" and "m3" are each an INTEGER constant, an INTEGER constant name or an INTEGER expression. An INTEGER expression may contain implied-DO variables of other implied-DO lists that have this implied-DO list within their ranges.

An iteration count is established from "m1", "m2" and "m3" exactly as it is for a DO loop (see Section 8.4).

Example:

```
      DATA a/4.2/,b(1)/5.4/
      DATA ( (x(j,i), i = 1,j), j = 1,5)
     &            /1,2,2,3*3,4*4,5*5/
      DATA j/4/, (c(j), j = 1,10)/10*2.0/
```

In the last statement, the value 4 is assigned to the variable "j". The "j" of "c(j)" is a dummy variable for use in the implied-DO.

7 ASSIGNMENT STATEMENTS

Execution of an assignment statement defines either a variable, array element or substring. The four kinds of assignment statements are described in this chapter:

> **numeric;**
> **logical;**
> **character;** and
> **statement label (ASSIGN).**

7.1 Numeric Assignment Statement

In numeric assignment statements, the value of a numeric expression is assigned to a variable or array element.

Syntax

```
v = e
```

Where:

"v" is a variable or an array element of numeric type; and

"e" is a numeric expression (see Section 4.3).

Semantic Rules

Execution of a numeric assignment statement causes the evaluation of the expression "e" by the rules in Section 4.1 and the assignment of the resultant value to "v".

Example:

```
i = x + 3 + SIN(y**2)
```

Arithmetic Conversion Rules

F77L recognizes the following numeric types: INTEGER*2, INTEGER*4, REAL*4, REAL*8 (DOUBLE PRECISION), COMPLEX*8 and COMPLEX*16. The order in which these numeric types are presented here is from low to high. The type of an expression is the highest type of the nonargument operands of the expression. To execute a numeric assignment statement, the expression is evaluated and its type converted to agree with the type of the assignment target. These type conversions can be viewed as a sequence of steps that transform a value from one type to the next to ultimately arrive at the required type.

For the purposes of the following descriptions, COMPLEX entities are considered to consist of two parts, "r" and "i", corresponding to the real and imaginary values.

INTEGER*2 = INTEGER

The low 16 bits of the INTEGER value are assigned to the INTEGER*2 datum. No guarantee is made to see that the INTEGER value is not too large. Note that this same assignment occurs for subscript and substring specifiers that are unsigned 16-bit integers.

INTEGER = INTEGER*2

The sign bit of the INTEGER*2 value is extended, and the result is stored in the INTEGER datum. If the result of the INTEGER*2 operation is greater than 32767, the INTEGER datum will not be correct.

INTEGER = REAL

The integer part of the REAL value is stored in the INTEGER datum. Any fractional value is ignored.

REAL = INTEGER

The INTEGER value is rounded and assigned to the REAL datum. Precision may be lost since only 24 bits of magnitude are stored. As a consequence the value may be rounded.

REAL = DOUBLE PRECISION

The DOUBLE PRECISION value is rounded and stored in the REAL datum. If the magnitude of the DOUBLE PRECISION value is smaller than 8.43E – 37, an arithmetic underflow is diagnosed. If the magnitude of the DOUBLE PRECISION value is greater than 3.37E38, an arithmetic overflow is diagnosed.

DOUBLE PRECISION = INTEGER

The INTEGER value is stored into the DOUBLE PRECISION datum. The precision is exact.

DOUBLE PRECISION = REAL

The REAL value is stored in the DOUBLE PRECISION value. The DOUBLE PRECISION datum contains all the precision of the REAL value.

DOUBLE PRECISION = COMPLEX

"r", a REAL value, is stored in the DOUBLE PRECISION datum. The DOUBLE PRECISION datum contains all the precision of the REAL value.

COMPLEX = DOUBLE PRECISION

The DOUBLE PRECISION value is rounded and stored in "r" and "i" is set to zero. If the magnitude of the DOUBLE PRECISION value is smaller than 8.43E – 37, an arithmetic underflow is diagnosed. If the magnitude of the DOUBLE PRECISION value is greater than 3.37E38, an arithmetic overflow is diagnosed.

COMPLEX = COMPLEX*16

The COMPLEX*16 "r" and "i" are rounded and assigned to the corresponding COMPLEX "r" and "i". If the magnitude of a COMPLEX*16 value is smaller than 8.43E – 37, an arithmetic underflow is diagnosed. If the magnitude of a COMPLEX*16 value is greater than 3.37E38, an arithmetic overflow is diagnosed.

COMPLEX*16 = COMPLEX

The COMPLEX "r" and "i" are assigned to the corresponding COMPLEX*16 "r" and "i". The COMPLEX*16 data contain all the precision of the COMPLEX data.

7.2 LOGICAL Assignment Statement

Syntax

```
v = e
```

Where:

"v" is a LOGICAL variable or a LOGICAL array element; and

"e" is a LOGICAL expression (see Section 4.6).

Semantic Rules

Execution of a LOGICAL assignment statement causes the evaluation of the LOGICAL expression "e" and the assignment and definition of "v" with the value of "e" (either ".TRUE." or ".FALSE.").

Example:

```
LOGICAL truth
truth = .FALSE.
```

7.3 CHARACTER Assignment Statement

Syntax

```
v = e
```

Where:

"v" is the name of a variable, an array element or a substring of type CHARACTER; and

"e" is a CHARACTER expression (see Section 4.4).

Semantic Rules

Execution of a CHARACTER assignment statement causes the evaluation of the expression "e" and the assignment and definition of "v" with the value of "e".

The expressions "v" and "e" may have different lengths. If "v" is longer than "e", blanks are concatenated to the right of "e". If "v" is shorter than the value of "e", the rightmost characters of "e" are truncated until "e" is the same length as "v". If "v" is a substring, only the character positions specified are defined; the character positions not specified are left unchanged.

Examples:

```
CHARACTER*6,c,d
c = 'ABCD'
d = 'AAAAAABB'
d(2:3) = 'BB'
PRINT *, c//d
```

prints

```
ABCDƀƀABBAAA
```

and

```
CHARACTER c*4, d*8
c = 'ABCD'
d = 'AAAAAABB'
d(2:3) = 'BB'
PRINT *, c//d
END
```

prints

```
ABCDABBAAABB
```

7.4 ASSIGN Statement

Syntax

```
ASSIGN s TO ivar
```

Where:

"s" is a statement label; and

"ivar" is an INTEGER variable.

Semantic Rules

Execution of an ASSIGN statement causes the statement label "s" to be assigned to the INTEGER variable "ivar".

The statement label must be the label of either an executable statement or a FORMAT statement in the same program unit as the ASSIGN statement.

An INTEGER variable defined with a statement label value may be redefined with the same or a different statement label value or with an INTEGER value. It is important to differentiate between the numeric assignment statement and the ASSIGN statement.

The statement:

```
i = 100
```

is a numeric assignment statement that stores the value 100 into INTEGER variable "i". The statement:

```
ASSIGN 100 TO i
```

is a statement label assignment statement that defines "i" to be the label value 100.

In the statement sequence:

```
10    CONTINUE
      ASSIGN 10 TO j
      i = j
      i = i + 1
```

neither of the latter two statements is defined in FORTRAN.

RESERVED FOR YOUR NOTES

8 CONTROL STATEMENTS

Control statements determine the execution sequence of statements in a program unit. FORTRAN defines two structures for controlling iteration or decision making: the DO structure and the IF...THEN...ELSE structure. Formal definitions of the two structures and related statements are provided in this chapter.

Iteration is provided by the DO statement structure. The DO loop uses a variable with a well-defined range of values to indicate how many times a group of statements (the DO range) should be executed.

Example:

```
        DO 5 i = 1,10,2
        PRINT *, a(i)
  5     CONTINUE
```

The IF...THEN...ELSE structure provides for alternative situations to a program loop.

Example:

```
  IF (i.LT.10) THEN
      PRINT *, a(i)
  ELSE IF (i.LT.20) THEN
      CALL warning
  ENDIF
```

Additional statements that control program flow include GO TO, arithmetic and logical IF, CONTINUE, PAUSE, CHAIN* and STOP. (The CALL and RETURN statements that control flow among subroutine and function subprograms are described in **Chapter 10**).

***NOTE:** The CHAIN statement is a feature supported in F77L and not in the extended-memory versions.

When using these control structures and statements, the following two rules must be observed:

1. A GO TO statement cannot transfer into a control structure. Control may be transferred within or out of a structure, but never into one.

2. Control structures may not overlap.

The following are illegal:

```
      DO 40, i=1,10
      IF(j.EQ.15)THEN
      j = j + 1
40    CONTINUE
      ELSE
      j = j – 1
      ENDIF
```

and

```
      IF(j.EQ.15)THEN
      DO 60,i=1,10
      j = j + 1
      ELSE
      j = j – 1
60    CONTINUE
      ENDIF
```

8.1 GO TO Statements

A GO TO statement transfers control to a statement identified by the GO TO. There are three types of GO TO statements:

the unconditional GO TO;
the computed GO TO; and
the assigned GO TO.

8.1.1 Unconditional GO TO

Execution of an unconditional GO TO statement transfers control so that the sequence of statements beginning with the statement label "s" is executed next. The syntax is:

```
GO TO s
```

Where:

"s" is a statement label.

Example:

```
a)          a = a + 2
b)          GO TO 400
c)          b = b + 2
d)   400    c = b + a
```

Execution of line "b" causes a transfer of control to the statement labeled 400. Note that line "c" is NEVER executed.

8.1.2 Computed GO TO

The computed GO TO statement selects one statement label from a list and transfers control to the executable statement identified by that label. The syntax is:

```
GO TO (s[,s]...)[,]e
```

Where:

"e" is a numeric expression; and each

"s" is the statement label of an executable statement in the same program unit as the computed GO TO statement.

Execution of a computed GO TO statement causes evaluation of the expression "e", which is then converted to an INTEGER value, INT(e). If the INTEGER value of "e" is "i", then the "ith" label in the list is selected and control is transferred to the statement identified by that label. If the value of "e" is less than

one or greater than the number of statement labels in the list of the computed GO TO, control is transferred to the next statement after the computed GO TO.

The same statement label may appear more than once in the computed GO TO statement.

Example:

```
GO TO (100,100,105,106) i
STOP
```

causes control to be transferred to the statement labeled 100 if "i" equals 1 or 2, to the statement labeled 105 if "i" equals 3, or to the statement labeled 106 if "i" equals 4. For all other values of "i", the STOP statement is executed.

8.1.3 Assigned GO TO

The assigned GO TO statement transfers control to a labeled statement according to the value stored in an INTEGER variable. The INTEGER variable must have been previously assigned a label value. The syntax is:

```
GO TO i [[,](s[,s...])]
```

Where:

"i" is an INTEGER variable; and

"s" is the statement label of an executable statement in the same program unit as the assigned GO TO statement.

At the time of execution of an assigned GO TO statement, the current value of "i" must have been defined by the prior execution of an ASSIGN statement (see Section 7.4).

The execution of the assigned GO TO statement causes a transfer of control to the executable statement identified by the most recent execution of such an ASSIGN statement.

Example:

```
INTEGER i
ASSIGN 100 TO i
GO TO i,(100,101,102)
```

causes control to be transferred to the statement labeled 100.

8.2 IF Statements

Single IF statements and the multiline IF...THEN...ELSE structure (described in the next section) transfer control to one of several alternative statements or groups of statements. There are two kinds of single IF statements:

arithmetic IF; and
logical IF.

8.2.1 Arithmetic IF

The syntax for the arithmetic IF statement is:

```
IF (e) [s1],[s2],[s3]

    or

IF (e) s1,s2,s3
```

Where:

"e" is a non-COMPLEX numeric expression; and

"s1", "s2" and "s3" are labels of executable statements in the same program unit as the arithmetic IF statement.

Execution of an arithmetic IF statement causes evaluation of the expression "e". If the value of "e" is negative, control is transferred to the statement identified by "s1". If the value of "e" is zero, control is transferred to "s2". If the value of "e" is positive, control is transferred to "s3". The same statement label may appear more than once in the same arithmetic IF statement.

If "s1", "s2" or "s3" is missing and "e" is evaluated to transfer control to the missing label, control is transferred to the next executable statement.

Example:

```
1       i = 10
2       IF(i)10,20,
3       i = j
```

causes line "3" to be executed next and

```
1       i = 0
2       IF(I)10,,20
3       i = j
```

causes line "3" to be executed next.

It should be noted that, since REAL and DOUBLE PRECISION values are approximations, the expression "e" may not evaluate exactly to the expected value.

8.2.2 Logical IF

The logical IF statement causes the conditional execution of one statement, depending upon the truth value of a LOGICAL expression. The syntax is:

```
IF (e) st
```

Where:

> "e" is a LOGICAL expression (see Sections 4.5 and 4.6); and
>
> "st" is any executable statement except a DO statement, another logical IF statement, a block-IF statement or an END statement.

Execution of a logical IF statement causes evaluation of the expression "e". If the value of "e" is true, statement "st" is executed. If the value of "e" is false, statement "st" is not executed, and the execution sequence continues with the next executable statement.

The execution of a function reference in the expression "e" of a logical IF statement is permitted to affect entities in the statement "st" (see **Evaluation of LOGICAL Expressions** in Section 4.6).

The statement:

```
IF (f(x,i) .EQ. 3) g(i) = 0.0
```

assigns the value of 0.0 to "g(i)", if "f(x,i)" is equal to 3.

8.3 Block-IF Structures

The block-IF allows conditional execution of one or more groups of statements. (The block-IF differs from the logical IF, where only one statement may be conditionally executed).

The syntax of the block-IF is:

```
IF (exp) THEN
    .
    .
    .
[ELSE [IF (exp) THEN
    .
    .
    .
[ELSE
    .
    .
    .]]]
ENDIF
```

Where:

"exp" is a LOGICAL expression.

8.3.1 Simple IF...ENDIF Structures

The IF...ENDIF form of the block-IF is more powerful than the logical IF statement because as many statements as desired may be included in the group between the IF statement and the ENDIF statement. The LOGICAL expression following the word "IF" is evaluated. If the value of the expression is true, then the statements between the IF and ENDIF are executed. If the value of the expression is false, then control is transferred to the first statement after the ENDIF.

NOTE: A good programming practice is to indent at each level of nesting to reduce errors and make programs more readable.

Example:

```
1  READ *, i, j
2  IF ( i.LT.j ) THEN
3      i = j
4      j = j + 10
5  ENDIF
6  m = i + j
```

If "i" is less than "j" after input, lines 3 and 4 are executed. If "i" is not less than "j" after input, line 6 is executed next, and lines 3 and 4 are not executed.

In a block-IF, there cannot be a statement following the LOGICAL expression or keyword THEN. If the above sequence had been written as follows:

```
1  READ *, i, j
2  IF ( i.LT.j ) THEN i = j
3      j = j + 10
4  ENDIF
5  m = i + j
```

line 2 would be treated as if it were a logical IF statement, and line 4 would be in error.

8.3.2 ELSE Statement

The ELSE statement causes a second group of statements to be executed if the LOGICAL expression in an IF...THEN...ELSE structure is false. The ELSE statement cannot be used outside an IF...THEN...ELSE structure.

The LOGICAL expression is evaluated. If its value is true, all statements between the IF and the ELSE statements are executed, after which control is transferred to the first statement after the ENDIF statement. If the value of the LOGICAL expression is false, control is transferred to the first statement after the ELSE statement, and all statements between the ELSE statement and the ENDIF statement are executed.

Example:

```
1  READ *,i,j
2  IF(i.LT.j)THEN
3      i = j
4      j = j + 10
5  ELSE
6      j = i
7      i = i + 10
8  ENDIF
9  m = i + j
```

If "i" is less than "j" after input, lines 3 and 4 are executed and control is transferred to line 9. If "i" is not less than "j", lines 6 and 7 are executed and line 9 is then executed.

8.3.3 ELSE IF Statement

The ELSE IF statement allows you to define several conditionally executable blocks of statements, rather than only two. If there is an IF (exp) clause immediately following the ELSE, the LOGICAL expression in the clause is evaluated if, and only if, the ELSE part of the IF...THEN...ELSE structure is executed.

Just as with the two-choice IF...ELSE, only one group of statements within an IF...ELSE...IF structure can be executed. Each LOGICAL expression is tested in turn until one is found to

be true. When a LOGICAL expression is true, the statements between that IF or ELSE IF and the next ELSE IF, ELSE or ENDIF are executed. After that group of statements is executed, control transfers to the first statement after the ENDIF, and any remaining ELSE IF or ELSE statements in the block IF are never executed. If none of the LOGICAL expressions are true and there is no simple ELSE statement at the end of the structure, then no group of statements within the block IF is executed.

```
1   READ *,i,j
2   IF(i.LT.j)THEN
3       i = i + 10
4   ELSE IF(i.EQ.j)THEN
5       j = j + i
6       i = j + i
7   ENDIF
8   m = i + j
```

If "i" is less than "j" after input, line 3 is executed and control is transferred to line 8. If "i" is not less than "j" after input, line 4 is executed, and the LOGICAL expression in the ELSE IF clause is evaluated. When "i" is equal to "j", lines 5 and 6 are executed, and control is transferred to line 8. If "i" is greater than "j" on input, control is transferred to line 8.

You may use as many ELSE IF clauses as desired:

```
1     READ *, i
2     IF (i.LT.10) THEN
3         j = i
4     ELSE IF (i.LT.100) THEN
5         j = i - 100
6     ELSE IF(i.LT.1000)THEN
7         j = i - 1000
8     ELSE IF(i.LT.10000)THEN
9         j = i - 10000
10    ENDIF
11    m = j
```

If "i" is less than 10000 on input, one of the lines between line 1 and 10 is executed. If "i" is not less than 10000 on input, control is transferred to line 11.

A simple ELSE statement may be used as the LAST choice before the ENDIF. The group of statements between the ELSE and ENDIF are executed whenever all of the preceding LOGICAL expressions are false.

Example:

```
1  READ *,i
2  IF(i.LT.10)THEN
3      j = i
4  ELSE
5      j = i - 100
6  ELSE IF(i.LT.1000)THEN
7      j = i - 100
8  ENDIF
9  m = j
```

This structure is illegal because the simple ELSE statement (line 4) comes before an ELSE IF (line 6). Therefore, the ELSE IF is meaningless and causes a FATAL compiler diagnostic.

<u>Nesting IF...THEN...ELSE Structures</u>

In addition to the ELSE IF clause, it is possible to nest IF...THEN...ELSE structures inside the THEN group or ELSE group of another IF...THEN...ELSE structure. A nested IF...THEN...ELSE must be totally contained within the THEN group or the ELSE group, and it may not overlap any other IF...THEN...ELSE groups.

Example:

```
1     PROGRAM main
2     READ *,i,j
3     IF(i.LT.j)THEN
4         i = j
5         IF(j.GT.100)THEN
6             j = 100
7         ELSE
8             j = 0
9         ENDIF
10    ELSE
11        i = j + 100
12    ENDIF
13    PRINT *,i,j
14    END
```

The IF...THEN...ELSE structure from lines 5 through 9 is nested inside the THEN group of the outer IF...THEN...ELSE structure from lines 3 through 12.

The following construction is NOT allowed:

```
1     PROGRAM main
2     READ *,i,j
3     IF(i.LT.j)THEN
4         i = j
5         IF(j.GT.100)THEN
6             j = 100
7         ELSE
8             j = 0
9         ELSE
10            i = j + 100
11        ENDIF
12    ENDIF
13    PRINT *,i,j
14    END
```

This construction is illegal because the nested IF...THEN...ELSE started on line 5 overlaps from the THEN group into the ELSE part of the outer IF...THEN...ELSE structure. The occurrence of the two ELSE statements in a row on lines 7 and 9 is illegal and a symptom of the illegal nesting.

Nesting Other Block Structures Within an IF...THEN...ELSE Structure

DO loops (see Section 8.4) may be nested within an IF...THEN...ELSE structure. As stated earlier, a nested block structure must be contained entirely within the THEN group or the ELSE group without overlapping. The ELSE or ENDIF statement may not be the terminal statement of a DO range.

8.4 DO Statement

DO constructs specify the repeated execution of a sequence of statements. The sequence of one or more executable statements is called the DO range. The range begins with the statement immediately following the DO statement and ends with the terminal statement. The variations of the DO statement and their functions are listed below.

The DO statement controls execution of the range and may include a DO label and/or a DO variable that is incremented and associated with how many times the loop is executed. The optional label is one way to designate the terminal statement.

The DO WHILE statement controls execution within the designated range based on the truth of a logical expression defined within the statement. DO WHILE also has an optional label to designate the terminal statement.

The END DO statement (which must not have a statement label) terminates either of the ranges when the DO statements do not specify a label.

The syntax for the DO statement is:

```
DO [s[,]][i=e1,e2[,e3]]
```

Where:

"s" is the label of an executable statement. The statement identified by "s", called the terminal statement of the DO loop, must physically follow and appear in the same program unit as the DO;

"i" is the name of a non-COMPLEX numeric variable called the DO variable and is the index of the loop;

"e1", "e2" and "e3" are non-COMPLEX numeric expressions:

"e1" defines the initial value of the index;

"e2" defines the limit value of the index; and

"e3" defines the increment of the index on each succeeding iteration. If "e3" is omitted, it is assumed to be one; "e3" must not be zero. After the loop begins execution, changing "e1", "e2" or "e3" has NO effect on the number of times the range is executed.

A DO statement causes the repeated execution of all statements in its range, beginning with the first executable statement after the DO statement through and including the statement labeled "s", the terminal statement. The index "i" is equal to "e1" on the first iteration, "e1" + "e3" on the second iteration and so on, until "i" cannot again be incremented by "e3" without exceeding the value of "e2"; i.e., if "i" + "e3" equals "e2", the range is executed one more time.

The value of "e3", the incremental value of the index, may be negative. If it is, the value of "e2", the limit value of the index, must be less than the value of "e1", the initial value of the index, to execute the loop at least once.

The terminal statement of a DO loop must not be an unconditional GO TO, arithmetic IF, IF...THEN...ELSE, ELSE, ELSE IF, ENDIF, RETURN, STOP, END or DO statement.

The index of the loop must not be changed (except by the increment of successive iterations) while the loop is active (i.e., while it is being executed). This means that the index must not appear as an input list item, must not be the index of another active DO loop, must not be redefined by a subprogram called during the execution of the loop and must not be assigned a value.

When the DO loop is satisfied, the index variable "i" retains the value it had during the last iteration of the loop plus "e3".

After the execution of the loop:

```
         DO 5 j = 1,5
         i = j
    5    CONTINUE
```

"j" has the value 6 and "i" has the value 5.

Control may be transferred outside a DO loop before it is satisfied; if it is, the index variable retains the value it had when the transfer was made. You may NOT transfer control into the range of a DO loop under any circumstances.

Note that the label is optional and an END DO would be the terminal statement of the range if, and only if, the label was not specified.

Example:

```
         DO i = 1, 5, 2
           j = i + 1
         END DO
```

If the following form is used:

```
         DO [label]
```

the range is repeatedly executed until a statement causes the loop to be exited or stopped.

Example:

```
         DO 10
           IF ( i .EQ. j ) STOP
           SUM = SUM + a(i, j)
           i = i + 1
    10   CONTINUE
```

8.4.1 DO WHILE Statement

DO WHILE statement iteration is controlled by a logical expression. The truth of the controlling logical expression is tested before loop execution. If the logical expression is true, the range is executed and control is returned to the top of the loop to test the controlling logical expression. If the logical expression is false, control is transferred to the first executable statement following the loop.

NOTE: A DO WHILE loop without a label requires an END DO statement for termination (see section 8.4.2).

The syntax of the DO WHILE statement is:

```
DO [label[,]] WHILE (e)
```

Where:

"label" is an executable statement label; and

"(e)" is a logical expression that controls the iterations.

Example:

```
      DO 20 WHILE (I .GT. O)
        PRINT*, I
      I = I - 1
20  CONTINUE
```

8.4.2 END DO Statement

The END DO statement terminates the execution of DO and DO WHILE statements which do not have labels. Note that END DO statements cannot have statement labels.

The syntax of the END DO statement is:

```
END DO
```

Example:

```
DO WHILE (I .GT. O)
  PRINT*, I
  I = I – 1
END DO
```

Nesting DO Loops

The range of any DO loop may contain one or more DO statements, defining loops that are said to be nested inside the range of the outer or containing DO loop. The range of a nested loop must lie entirely within the range of its containing loop. A nested loop must not use the index of its containing loop as its own index, although it may use the index of its containing loop as a variable in its own loop or in "e1", "e2" or "e3".

More than one DO loop may have the same terminal statement.

Examples of DO Loops

The following examples compute k factorial:

```
      kfact = 1
      DO   10 i = 2,k
         kfact = kfact*i
10    CONTINUE

      kfact = 1
      DO 10 i = 2,k
10    kfact = kfact*i
```

The following is an example of using a negative index:

```
      DO 10  i = 4,–5,–1
         j = i
10    CONTINUE
```

After the loop is exhausted, "j" equals –5 and "i" equals –6.

Here is an example of using the incremental value:

```
      DO 10  i = 2,5,2
         j = i + 1
10    CONTINUE
```

On the first iteration of this loop, "i" equals 2 and "j" equals 3. On the second iteration of this loop, "i" equals 4 and "j" equals 5. There is no third iteration.

Here is an example of nested DO loops:

```
      n = 0
      DO 20  i = 1,10
         j = i
         DO 10  k = 1,5
            m = k
10       n = n + 1
20    CONTINUE
```

After execution of both loops, "i" equals 11, "j" equals 10, "k" equals 6, "m" equals 5 and "n" equals 50.

<u>Nesting Other Block Structures Within a DO Loop</u>

An IF...THEN...ELSE structure may be nested within a DO loop. A nested block structure must be contained entirely within the range of the DO loop. For the IF...THEN...ELSE structure, the IF statement must follow the DO statement, and the ENDIF statement must precede the DO loop terminal statement.

The following examples are illegal:

```
      DO 40, i = 1,10
         IF(j.EQ.15) THEN
            j = j + 1
40    CONTINUE
         ELSE
            j = j – 1
      ENDIF
```

```
        IF(j.EQ.15) THEN
           DO 60,i = 1,10
              j = j + 1
             ELSE
           j = j - 1
60      CONTINUE
           ENDIF
```

8.5 CONTINUE Statement

The CONTINUE statement has no effect in FORTRAN; the statement may be used simply for program clarity. The syntax is:

```
[CONTINUE]
```

8.6 PAUSE Statement

The PAUSE statement causes a temporary suspension of the execution of a program. The syntax is:

```
PAUSE [n]
```

Where:

"n" is optional and is either a CHARACTER or INTEGER constant.

When a PAUSE statement is encountered, program execution stops, and the constant (if specified) and the text,

"Press Enter to continue"

are printed at the console. To resume program execution, press the **ENTER** key.

Here are two examples of PAUSE statements:

```
10010       PAUSE "Pause at statement labeled
          &10010"
            PAUSE "Insert floppy"
```

8.7 CHAIN Statement

The CHAIN* statement halts execution of the current program and begins execution of the specified program.

***NOTE:** The CHAIN statement is a feature supported in F77L and not in the extended-memory versions.

The syntax is:

```
CHAIN pname [,cvar]
```

Where:

"pname" is a CHARACTER expression specifying the name of the program file (which must be an "executable" file); and

"cvar" is a CHARACTER expression that is to be passed to the CHAINed-to program as if "cvar" were typed on the command line (see **GETCL** in **Chapter 12 – DOS Interface Subroutines**).

All open files and blank common are passed to the CHAINed-to program. If the sizes of blank common disagree, only the lesser number of words are passed to the CHAINed-to program.

8.8 STOP Statement

The STOP statement terminates program execution. The syntax is:

```
STOP [n]
```

Where:

"n" is optional and is either a CHARACTER constant or an INTEGER constant.

When a STOP statement is executed, the optional constant is displayed on the console.

INPUT/OUTPUT STATEMENTS 9

INPUT/OUTPUT STATEMENTS 9

9 INPUT/OUTPUT STATEMENTS

This chapter contains detailed input/output (I/O) and file organization information. For console input/output, refer to Section 9.5 where the INPUT, READ and PRINT statements are presented.

The following statements are used to manipulate files and to perform file input/output:

BACKSPACE	**INQUIRE**	**REWIND**
CLOSE	**OPEN**	**WRITE**
ENDFILE	**READ**	

To use these statements as many as four categories of information may be required:

1. The file organization of the input or output file: file organization refers to the manner in which data are read from or written to a file. File organization is specified in the OPEN statement.

2. The statement specifiers: all input/output statements (except the console input/output statements: INPUT, READ and PRINT) share the same syntax for these input/output specifiers.

3. The edit descriptors of the input/output data specify the number of digits on each side of the decimal point, spaces to skip, etc. Edit descriptors may be specified by FORMAT statements or by list-directed formatting.

4. The variables, arrays and expressions within the program to be used in performing data transfers: these items are given in the input/output lists of READ, WRITE, INPUT and PRINT statements. The order of the items in the input/output list is the order in which they are input or output.

9.1 Input/Output Specifiers

For input/output specifiers, most statements share the following syntax:

keywrd (specifier=item, specifier=item,...)

Where:

"keywrd" is OPEN, INQUIRE, READ, WRITE, REWIND, ENDFILE, BACKSPACE or CLOSE;

"specifier" may be one of the several described below and in Sections 9.3, 9.4 and 9.6 through 9.10 (for example, ACCESS=, NAME=, and FMT=), depending upon the keyword; and

"item" is an expression, variable, array element, label or "*", depending upon the specifier.

The syntax "specifier=item" allows the specifiers to be given in any order. If the first specifier in an input/output statement is the unit number, then the "UNIT=" text is optional. The statements:

```
WRITE (UNIT=5, ERR=10)

        and

WRITE (5, ERR=10)
```

are equivalent. The text for the FILE= or FMT= specifier may also be omitted if one of them appears second following a unit specifier in which the UNIT= text is also omitted. (FILE= and FMT= can never appear together in the same statement; see the descriptions of these specifiers in Sections 9.3 and 9.4.)

In general, input/output statements require the UNIT= specifier to identify where the input/output is to be performed. Although not required, the ERR= and IOSTAT= specifiers may also be used with most input/output statements. Because their syntax and function are the same for all input/output statements, these

three specifiers are described below. The remaining specifiers are described with the input/output statement(s) with which they are used.

UNIT= Numeric Expression or *

This specifier must be used with OPEN, READ, WRITE, CLOSE, REWIND, BACKSPACE and ENDFILE statements. It may be used with the INQUIRE statement (see Section 9.10).

The numeric expression defines the unit number to be assigned to the file. The maximum value for a unit number is 32767. Unit * is reserved to designate the console.

A unit must not be connected to more than one file at the same time. The file with which data are being exchanged is designated by a READ or WRITE statement using the file's unit number.

The text "UNIT=" is optional if this specifier is first in the control list. The following are equivalent:

```
OPEN (UNIT=3)
OPEN (3)
```

In this example, a scratch file is opened for unit 3 because no file has been specified.

ERR= Statement Label

This specifier may be used with OPEN, READ, WRITE, CLOSE, INQUIRE, REWIND, BACKSPACE and ENDFILE statements. The label specifies where control is to be transferred if statement execution is not completed because of an error condition, such as specifying a nonexistent file on an OPEN with "OLD" status. For example, the following statement continues execution at the statement labeled 10 if "joe" is not in the directory:

```
OPEN (5, ERR=10, FILE='joe', STATUS='OLD')
```

IOSTAT= INTEGER Variable or Array Element

This specifier may be used with OPEN, READ, CLOSE, WRITE, INQUIRE, REWIND, BACKSPACE and ENDFILE statements (see Section 9.3).

If the IOSTAT specifier is present and an error occurs, the INTEGER variable or array element designated is set in accordance with the IOSTAT error code table in **Appendix L**. If both IOSTAT= and ERR= specifiers are present, the IOSTAT variable is set before the branch to the statement designated by the ERR= specifier; if no ERR= specifier is present, execution continues sequentially. If neither IOSTAT= nor ERR= specifiers are present, an input/output error stops execution after an error message is printed to the console (STDERR).

9.2 File Concepts

A file is a sequence of records. There are two kinds of files: internal and external. An internal file is allocated and exists only within a program unit; it is used to edit data with the standard READ and WRITE statements. An external file may be a disk (or other similar media) file, which can be saved for future use and may be used by any number of programs, or it may be a non-disk device accessed by a DOS reserved name.

Internal Files

Internal files provide a means of reading and writing data to and from internal storage (CHARACTER variables, CHARACTER array elements or CHARACTER substrings) using formatted READ or WRITE statements. A CHARACTER-array internal file may be thought of as a formatted, sequential file, having one record for each array element. All other internal files may be thought of as one-record, formatted files.

The syntax for internal READ and WRITE statements is as follows:

```
READ ([UNIT=]var, [FMT=]fmt [,ERR=s][,END=s]) iolist

WRITE ([UNIT=]var, [FMT=]fmt [,ERR=s][,END=s]) iolist
```

Where:

"var" is the symbolic name of a CHARACTER variable, CHARACTER array, CHARACTER array element or CHARACTER substring;

"fmt" is the label of a FORMAT statement, a CHARACTER expression that yields a format specification or the list-directed input/output designator *;

"s" is the label of an executable statement to which control is transferred if an error occurs when the ERR= specifier is used, or if the end-of-file condition occurs when the END= specifier is used; and

"iolist" is composed of variable names, implied-DO loops, array element names, array names and substring names. On output, "iolist" may also contain expressions whose values are to be output.

The only input/output statements that may reference internal files are READ and WRITE. A record of an internal file is a CHARACTER variable, CHARACTER array element or CHARACTER substring. If the file is a CHARACTER variable, CHARACTER array element or CHARACTER substring, it has only one record whose length is the same as the length of the variable, array element or substring. If the file is a CHARACTER array, it has one record for each array element. The ordering of records in a CHARACTER array corresponds to the ordering of elements in an array. Every record of the file has the same length, which is the length of an array element of that array.

An Internal file record may be read only if it has been assigned a value previously. A CHARACTER variable or array element or a substring that is an internal file record may also appear in other contexts (e.g., CHARACTER assignment statements).

An internal file is always positioned at the initial point of the first record before each READ or WRITE using the internal list.

Examples:

```
          CHARACTER*80,buffer /'ƀƀ1ƀƀ2ƀƀ3'/
          READ(buffer,20)i,j,k
   20     FORMAT(3i3)
          PRINT 20, i, j, k
          END
```

prints

```
   1   2   3
```

and

```
   CHARACTER*1 zero /'0'/
   CHARACTER*1 one
   INTEGER acumultr
   READ (zero,'(i1)') acumultr
   acumultr = acumultr + 1
   WRITE (one,'(i1)') acumultr
   PRINT *, one
   END
```

prints

```
   1
```

External Files

External files may be disk files or non-disk devices accessed using DOS reserved filenames. External files are categorized by their method of access; they may be accessed (read or written) by either of two methods: sequential or random. Records in a sequential file are accessed in the order in which they were written.

Records in a random (or direct) file are identified by unique record numbers, starting with one, and may be accessed in any order.

Characteristics of F77L File Types		
	Formatted	**Unformatted**
Sequential	ASCII Characters. Terminated by a carriage return/line feed. Variable-length records. Can be used with DOS devices. or disk files. File can be printed or displayed easily. Records must be processed in order. Usually slowest.	Binary Data. Separated by record marker. Variable-length records. Disk files only. Not easily processed outside of F77L. Records must be processed in order. Faster than formatted.
Direct	ASCII Characters. Fixed-length records. Record 0 is a header. Disk files only. Not easily processed outside of F77L.* Records can be accessed in any order. Same speed as formatted sequential disk files.	Binary Data. Fixed-length records. Record 0 is a header. Disk files only. Not easily processed outside of F77L.* Records can be accessed in any order. Fastest I/O.
	Transparent	
	Binary Data: ASCII I/O via CHARACTER data type. No record marker or headers. Can be used with DOS devices or disk files.	Good for files that interact with non-F77L programs. Records can be processed in any order. Very fast.

NOTE: *Refer to the "/D" Compiler Option in **Appendix A**.

Either ASCII or binary data may be read from or written to a file; however, the entire file must be either binary or ASCII, not both. A binary file is referred to as "UNFORMATTED".

Any external file may also be accessed as "TRANSPARENT", which is a special version of UNFORMATTED DIRECT. The "record length" is 1 byte, but normal FORTRAN restrictions on end-of-record do not apply. An input/output transfer begins at the byte number specified in the REC= specifier in a READ or WRITE statement, or at the byte following the byte at which the last transfer ended, if REC= is not specified. The number of bytes transferred is equal to the total length of items in the input/output list.

A special type of external file is a non-disk device, which is referred to as a device. A device is either the console (unit *) or a unit opened with one of the DOS reserved filenames:

CON, AUX, COM1, COM2, LPT1, LPT2, LPT3, PRN or NUL

specified for FILE=.

These devices can be opened as FORMATTED SEQUENTIAL or TRANSPARENT files. These files differ from disk files in that BACKSPACE, REWIND, ENDFILE and REC= in a READ or WRITE statement cannot be used with them. Also, the output sent to them cannot be saved and read later as with a disk file.

NOTE: The DOS device drivers for devices other than the console (CON) do not buffer input, so the program must be waiting when input arrives, and data arriving at a fast rate may not be read correctly.

If a device is opened as FORMATTED SEQUENTIAL, the first character of a fixed-format output record is used for carriage control (see Section 9.6), unless overridden with CARRIAGE CONTROL= in the OPEN statement.

If a device is opened as TRANSPARENT, bytes are read from and written to the device sequentially without any conversion, echoing or added control characters. A single byte can be read from a device (including the console), without a carriage return

(**ENTER**) being pressed. In most cases, only CHARACTER data should be read from or written to a device opened as TRANSPARENT.

Preconnected Units

Unit numbers 5 and 6 do not need to appear in an OPEN statement before being used in an input/output statement. Unit 5 is automatically connected to STDIN (the console keyboard), and unit 6 is automatically connected to STDOUT (the console screen). For more information about STDIN, STDOUT and redirecting input and output, see your DOS Manual. Units 5 and 6 may also be explicitly opened to a file and will then function like any other opened unit.

9.3 OPEN Statement

An OPEN statement creates a new file, reopens an old file or changes one of the characteristics of an opened file. The syntax for this statement is:

OPEN (spec=sval [,spec=sval]...)

Where:

"spec" is one of UNIT, FILE, FORM, STATUS, ACCESS, IOSTAT, BLANK, RECL, ERR,

CARRIAGE CONTROL, POSITION, ACTION, DELIM, PAD or BLOCKSIZE.

The specifiers may be in any order.

"sval" is either:

1. an expression: either positive numeric or CHARACTER, depending on the "spec" used;
2. an INTEGER variable or array element for IOSTAT; or
3. a statement label for ERR.

The keywords "UNIT=" and "FILE=" are optional if they are the first and second specifiers to appear. Note that with or without UNIT=, a unit number must be specified.

The UNIT=, ERR= and IOSTAT= specifiers are described in Section 9.1. The remaining specifiers are described below.

Specifying UNIT=* in the OPEN statement allows modifying BLANK=, CARRIAGE CONTROL=, PAD=, DELIM=, BLOCKSIZE= and RECL= for the console input/ouput.

<u>FILE= CHARACTER Expression</u>

The CHARACTER expression identifies the name of the file and is a sequence of characters naming a file in the operating system. Standard file naming conventions are used. The text "FILE=" is optional if this specifier is second and the unit is specified first without "UNIT=". The following examples are equivalent:

```
    OPEN (UNIT=5,FILE='table')

        and

    OPEN (5,'table')
```

<u>STATUS= CHARACTER Expression</u>

The valid options for STATUS= are "NEW", "OLD", "UNKNOWN" or "SCRATCH". The default status is "UNKNOWN". To guarantee that a file didn't exist previously, specify STATUS="NEW". The OPEN statement will report an error if a file by that name already exists. To guarantee that a file already exists, specify STATUS= "OLD". If the existence of a file is uncertain, specify STATUS= "UNKNOWN" (or omit the STATUS specifier, since the default is "UNKNOWN"). If the file does exist, it is opened; if it does not exist, it is created.

STATUS="SCRATCH" specifies a temporary file available only during execution of the current program. Because "SCRATCH" creates a temporary file, do not use the FILE= specifier with that status.

Examples:

```
OPEN(UNIT=2, STATUS='SCRATCH')
OPEN(5,'table', STATUS='NEW')
OPEN(j + 1,'file4', STATUS='OLD')
```

ACCESS= CHARACTER Expression

F77L supports two file access methods: sequential and random. Records in a sequential file are accessed in the order in which they were written. Records in a random (or direct) file are identified by unique record numbers starting with one, and may be accessed in any order, including sequentially. Either ASCII or binary data may be written to a file (see FORM=, below); however, the entire file must consist of either ASCII or binary data, not both.

The expressions that may be used with the ACCESS= specifier in an OPEN statement are:

SEQUENTIAL denotes a sequential file, either formatted or unformatted. This is the default access if none is specified.

APPEND denotes a sequential file like "SEQUENTIAL", but positions the file for writing at the position following its current last record.

TRANSPARENT denotes a random access, unformatted file with a record length of 1 byte and no header.

DIRECT denotes a random access file, either formatted or unformatted.

Examples:

```
  OPEN(1,'table',ACCESS='SEQUENTIAL',
& STATUS= 'NEW')
  OPEN(2, STATUS='SCRATCH', RECL=32,
& ACCESS= 'DIRECT')
```

FORM= CHARACTER Expression

This specifier indicates the file format and has two possible options: "FORMATTED" and "UNFORMATTED". If "FORMATTED" is specified, then the file will contain only ASCII characters. If "UNFORMATTED" is specified, then the file records contain binary data. ASCII data may be placed in an unformatted file by writing CHARACTER data. The default form for a sequential file is "FORMATTED". For a direct file, the default is "UNFORMATTED".

Example:

```
      OPEN(UNIT=1,STATUS='NEW',FILE='table',
     &ACCESS='DIRECT',FORM='FORMATTED',RECL=5)
```

RECL= Numeric Expression

RECL (record length), the number of bytes in a record, must be specified when creating a direct file.

For a FORMATTED SEQUENTIAL file, RECL sets the maximum line length for list-directed output. One line of console input or output normally contains 80 characters.

If FORM="FORMATTED", RECL=80, and list-directed output is used, then a line in this file would contain a maximum of 80 characters, not including the carriage return and line feed.

Refer to **Appendix N** for the maximum allowable record lengths.

BLANK= CHARACTER Expression

There are two possible options for this specifier: "NULL" and "ZERO". If it is "NULL", then all blanks in numeric-formatted input fields are ignored except that a field of all blanks is input as one zero. If "ZERO" is specified, then all but leading blanks are treated as if they were zeros. The default option is "NULL".

CARRIAGE CONTROL= CHARACTER Expression

There are two possible options for this specifier: "LIST" and "FORTRAN". If the CHARACTER expression is "FORTRAN", then fixed format sequential output to the indicated unit will be "printed" (see **Carriage Control** in Section 9.6). If you select "LIST", then the entire formatted sequential record will be output, followed by a carriage return and a line feed. DOS devices (see External Files in this section) default to "FORTRAN"; disk files default to "LIST".

POSITION= CHARACTER Expression

There are three possible options for this specifier: "ASIS", "REWIND" and "APPEND". The connection must be for sequential access. A file that did not previously exist (a new file, either specified explicitly or by default) is positioned at its initial point. "REWIND" positions an existing file at its initial point. "APPEND" positions an existing file such that the endfile record is the next record, if it has one. "ASIS" leaves the position unchanged if the file exists and is already connected. "ASIS" leaves the position unspecified if the file exists but is not connected. If this specifier is omitted, the default value is "ASIS".

ACTION= CHARACTER Expression

The ACTION= parameter in the OPEN statement allows the specification of sharing modes. The syntax is:

    action = character

Where:

    the value of "character" is arg[,arg]

and the possible values for arg have the following meanings:

Access Modes:

READ	Only reading allowed on the file
WRITE	Only writing allowed on the file
READ/WRITE	Reading or writing allowed

Sharing Modes:

DENYBOTH	For exclusive use by this unit in this process
DENYWRITE	Allows reading by others, but not writing
DENYREAD	Allows writing by others, but not reading
DENYNONE	Allows others to open the file for reading or writing

Any file to be shared must be opened with a sharing mode specified by all programs that want to access it.

Example:

```
OPEN (UNIT=9, FILE='G:SHARE.IT',
ACTION='WRITE,DENYWRITE', ERR=100)
```

UNIT 9 will be opened to have exclusive rights to write to the file. If another process has the file open with write permission or with DENYWRITE or DENYBOTH, the ERR= transfer to statement 100 will occur.

DELIM= CHARACTER Expression

There are three options for this specifier: "APOSTROPHE", "QUOTE" and "NONE". If "APOSTROPHE is specified, the apostrophe will be used to delimit CHARACTER constants written in list-directed or namelist formatting and all internal apostrophes will be doubled. If "QUOTE" is specified, the quotation mark will be used to delimit CHARACTER constants written in list-directed or namelist formatting and all internal quotation marks will be doubled. If the value of this specifier is "NONE", a character constant when written will not be delimited by apostrophes or quotation marks, nor will any internal apostrophes or quotation marks be doubled. If this specifier is omitted, the default value is "NONE". This specifier is permitted only for a file being connected for formatted I/O. This specifier is ignored during input of a formatted record.

PAD= CHARACTER Expression

There are two options for this specifier: "YES" and "NO". If "YES" is specified, a formatted input record is logically padded with blanks when an input list is specified and the format specification requires more data from a record than the record contains. If "NO" is specified, the input list and the format specification must not require more characters from a record than the record contains. If this specifier is omitted, the default value is "YES".

BLOCKSIZE= INTEGER Expression

In an OPEN statement, you can use the specifier BLOCKSIZE= to specify the size (in bytes) of the I/O buffer. The Default BLOCKSIZEs are listed in **Appendix N**.

Example:

```
OPEN (UNIT=1,FILE="FOO",BLOCKSIZE=4096)
```

The effectiveness of different I/O buffer sizes is both hardware and program dependent. The hardware dependencies relate to the speed of your CPU, NDP and hard disk performance. The program dependencies relate to the type of file access used.

9.4 READ and WRITE Statements

READ and WRITE statements are used to transfer data between the program and a file. The syntax for these statements is:

```
READ (spec=sval [,spec=sval]...)iolist
WRITE (spec=sval [,spec=sval]...)iolist
```

Where:

"spec" is one of ERR, UNIT, FMT, REC, IOSTAT or END;

"sval" is a numeric expression, CHARACTER expression or a statement label, depending on the "spec" used; and

"iolist" is composed of variable names, implied-DO loops, array element names, array names and CHARACTER substring names. On output "iolist" may also contain expressions whose values are to be output.

Examples:

```
READ(UNIT=1)x,y,z
WRITE(unitnum,FMT=10)x,y,z,x + y + z
```

The UNIT=, ERR= and IOSTAT= specifiers are described in Section 9.1. The remaining specifiers are described below.

<u>END= Statement Label</u>

The statement label identifies an executable statement to which control is transferred if an end-of-file condition is encountered during the execution of a READ statement. If an end-of-file condition occurs during the execution of an input statement that does not contain END=, ERR=, or IOSTAT= specifiers, execution of the program is terminated with an error message.

<u>FMT= Statement Label, CHARACTER Expression, CHARACTER Array Name, Assigned Variable or *</u>

The statement label specifies a FORMAT statement. If a statement label is present, the data are input or output according to the editing information supplied in the FORMAT statement (see Section 9.7). FMT= may also be followed by a CHARACTER expression or array which must contain the same information that would be in a FORMAT statement (except for the keyword FORMAT). An "*" specifies list-directed input/output (see Section 9.6).

A format must be specified for formatted input/output and must not be specified for unformatted input/output.

If the format option is an array name, the length of the format specification is considered to be a concatenation of all the array elements of the array in the order given by array element ordering (see **Array Element Storage** in Section 3.2). However, if the name of a CHARACTER array element is specified as the format option, the length of the format specification must not exceed the length of the array element.

If an INTEGER variable is the format option, it must have been previously assigned a label which is associated with a FORMAT statement.

NOTE: The text "FMT=" is optional if this specification appears second and the unit is specified first without "UNIT=".

Example:

```
WRITE(5,*)a + b
READ(5,FMT='(2F10.0,I5)')a,b,i
```

The parentheses must be included in the FORMAT character constant.

Example:

```
CHARACTER*4,a(10)
a(1)='(i5)'
a(2)='(i6)'
a(3)='(i7)'
READ (UNIT=4,FMT=a(1))i
READ (4,a(2))j
READ (FMT=a(3),UNIT=4)k
```

REC= Numeric Expression

The positive numeric expression specifies the record number in a random-access (direct) file to be read or written. If REC is not specified for a direct file, the file is accessed sequentially.

Iolist

The "iolist" specifies the entities whose values are transferred by an input/output statement. If an array name appears as an "iolist" item, it is treated as if all the elements of the array were

specified in the order given by the array element ordering. The following statements use the console to read into and then write from the ten elements of the REAL array "a":

```
      REAL a(10)
      READ (UNIT=*,FMT=10) a
      WRITE (*,11) a
10    FORMAT(10F10.6)
11    FORMAT(' '10F10.6)
```

An implied-DO list may be used to input or output multiple elements of an array (see Section 6.10).

The following statements:

```
      REAL a(10)
      ...
      READ (4,FMT=10)(a(i),i=3,5)
10    FORMAT (3F10.6)
```

cause values to be read from unit 4 into elements 3, 4 and 5 of "a".

9.5 Console Input/Output Statements

INPUT, PRINT and the short version of the READ statement are special cases of the READ and WRITE statements and are used to transfer data to and from the console only. The discussion presented here is a simplified one, showing by example how to do input and output using the console. The syntax for the statements are:

```
READ    [f]  [,iolist]
PRINT   [f]  [,iolist]
INPUT   [f]  [,iolist]

      or

READ    f [,iolist]
PRINT   f [,iolist]
```

Where:

"f" is the label of a FORMAT statement, a CHARACTER expression that yields a format specification, or *; and

"iolist" is an input or output list.

The INPUT and PRINT statements refer only to the console. They treat the console as a sequential, formatted file and do not require an OPEN statement.

The INPUT and PRINT statements allow a format specifier "f" that specifies explicit formatting. If the format specifier is an asterisk (*), list-directed formatting is used.

Unformatted (binary) data cannot be transferred to or from the console.

Here is an example that illustrates list-directed formatting:

```
PRINT *,'Please input a real number '
READ *,a
PRINT *,'The square of ',a,' is ',a**2
END
```

Each list-directed PRINT statement advances the console to a new line.

9.6 Input/Output Formatting

Formatting in input/output statements consists of specifying editing and data arrangement information. This information may be explicitly supplied by referencing a FORMAT statement or may be implicitly stated by using list-directed input/output in which data are edited and arranged using a predetermined format.

List-directed Input and Output

List-directed input/output is specified by an asterisk (*) as the format option or by omitting the format option in the short console input/output statements. The statements:

```
READ *, a
PRINT *, x,y,z
READ *, x(1),b,c
READ (UNIT=1,FMT=*)i,j,k
```

all use list direction.

For list-directed output, the editing performed depends on the type of the value to be output. The following table shows the editing format used for each data type:

Data Type	Editing
INTEGER	I12
INTEGER*2	I7
REAL	G15.6E2
DOUBLE PRECISION	G25.15D3
COMPLEX	' (',G15.6E2,',',G15.6E2,')'
COMPLEX*16	(',G25.15D3,',',G25.15D3,')'
LOGICAL	L2

NOTE: For COMPLEX, no leading or trailing blanks are output as part of the G specification.

CHARACTER values are printed in a field as wide as the length of the datum; i.e., no leading or trailing blanks are output unless they are included in the value.

Data prepared for list-directed input may be written without regard for field boundaries. A list-directed input record consists of values and value separators. A value is either a constant, a null value or has one of the forms: "r*c" or "r*". The form "r*c" specifies "r" successive appearances of the constant "c", and the form "r*" is equivalent to "r" successive null values. Note that the repeat count feature is limited to sequential file input.

The value separators are a comma (optionally followed by one or more blanks) or one or more contiguous blanks between two constants. A carriage return is treated like a blank, except in a character string enclosed in apostrophes or quotation marks, where it is ignored.

A slash encountered as a value separator during execution of a list-directed input statement causes termination of that input statement after the assignment of the previous value. If there are additional items in the input list, the effect is as if null values had been supplied for them.

Examples of list-directed data:

Type	Allowable	
INTEGER	2 –101 5	
REAL	3 54.77 –34.66E12 +9.99D + 23	
DOUBLE PRECISION	3 1.0D – 29	.123456789999D0 5.9
COMPLEX	(3, 5.)	(1.0D4, 3E6)
LOGICAL	TRUE T .F .TRUE.	
CHARACTER	ABC "don't" 'Use a ","'	'ABC DEF' 'don''t'

The syntax of a CHARACTER constant is a sequence of one or more ASCII characters enclosed in apostrophes. These delimiting apostrophes are not part of the datum represented by the constant. An apostrophe within a CHARACTER constant delimited by apostrophes must be represented by two consecutive apostrophes with no intervening blanks.

F77L allows use of quotation marks as delimiters in addition to apostrophes. A quotation mark within a CHARACTER constant delimited by quotation marks is represented by two consecutive quotation marks with no intervening blanks.

Carriage Control

By default, sending fixed-format formatted output to any of the following devices:

CON, LPT1, LPT2, LPT3, PRN, COM1, COM2 and AUX

is called "printing". Output to disk files is by default not "printed".

To change the defaults, see CARRIAGE CONTROL= in the OPEN statement (Section 9.3).

Printing means that the first character of the formatted sequential output record is not output and has the following effects:

Character	Action
0	double space
1	form feed
2	print 2 blank lines
3	print 3 blank lines
4	print 4 blank lines
5	print 5 blank lines
6	print 6 blank lines
7	print 7 blank lines
8	print 8 blank lines
9	print 9 blank lines
+	overprint previous line
&	append to previous line
any other	single space

The indicated action takes place before the line is printed and occurs whenever there is an attempt to place a character in the first print column, no matter where in a format such an attempt may take place. Carriage control is not available with list-directed output.

9.7 FORMAT Statement

The FORMAT statement is a nonexecutable FORTRAN statement that provides the program with editing and data arrangement information. The FORMAT statement must have a label (see Section 1.4.1). The syntax of the statement is:

FORMAT ([flist])

Where:

"flist" is a format list. The form of each list item of "flist" must be one of the following:

ned
[r]ed
[r](flist)

"ned" is a nonrepeatable edit descriptor;

"ed" is a repeatable edit descriptor; and

"r" is a nonzero, unsigned, INTEGER constant called the repeat specification.

The forms of repeatable edit descriptor "ed" are:

A	**Dw.d**	**Dw.dDe**	**Dw.dEe**
Aw	**Ew.d**	**Ew.dDe**	**Ew.dEe**
Iw	**Fw.d**	**Gw.dDe**	**Gw.dEe**
Iw.m	**Gw.d**	**Lw**	**Zw**

Where:

I, F, E, D, G, L, Z and A indicate the type of editing;

"d" is an unsigned INTEGER constant, specifying the fractional part;

"e" is a nonzero, unsigned, INTEGER constant, specifying the exponent field width;

"m" is an unsigned INTEGER constant, specifying the minimum digits; and

"w" is a nonzero, unsigned, INTEGER constant, specifying the field width.

The forms of a nonrepeatable edit descriptor are:

'aaaaa...'	**bX X**	**BN BZ**
"aaaaa..."	**:**	**/**
nHaaaaaa	**S SP SS**	**kP**
	Tc TLc TRc	

Where:

' (apostrophe), " (quotation mark), / (slash), : (colon), H, T, X, S, SS, SP, P, BN, BZ, TR and TL indicate the type of editing;

"a" is an ASCII character;

"n", "b" and "c" are nonzero, unsigned, INTEGER constants; and

"k" is an optionally signed INTEGER constant.

The comma used to separate list items in the "flist" may be omitted in the following cases:

following a P edit descriptor;
before or after a slash edit descriptor;
before or after a colon edit descriptor; or
following ", ', nH and X descriptors.

Symbolic names of constants must not be part of format specifications.

Interaction with Input/Output List

Input or output under control of a format depends jointly on the format specifiers and the input/output list items. A format list is processed from left to right, except as modified by a repeat specification on an edit descriptor or format list. Each input/output list item must have one corresponding repeatable

edit descriptor (except for COMPLEX items, which must have two). Nonrepeatable edit descriptors communicate directly with the record and do not correspond to input/output list items.

The form FORMAT(), where the input/output list is empty, may be used to cause an input record to be skipped or one empty output record to be written.

Processing of a format terminates when all the input/output list items have been processed, and one of the following is encountered: the last right parenthesis, a colon (:) edit descriptor or a repeatable edit descriptor.

If the last right parenthesis is encountered before all of the input/output list items have been processed, the file is positioned at the beginning of the next record and format <u>reversion</u> occurs. This means that format processing reverts to the beginning of the format list terminated by the last preceding right parenthesis, or if there is no preceding right parenthesis, it reverts to the first left parenthesis. If control reverts to a format list preceded by a repeat specification, the repeat specification is reused. Format reversion, of itself, has no effect on the established scale factor (P edit descriptor), plus sign control (S, SS, SP) or blank control (BN, BZ).

If a format list requires more input characters than a record contains, the record is padded by default on the end with enough blanks to satisfy the format list. This allows FORMATTED SEQUENTIAL records to be smaller, because trailing blanks are not required. Refer to the PAD= specifier in Section 9.10 for information on how to force formatted records to be large enough.

9.7.1 CHARACTER Constant Editing

The apostrophe or quotation mark edit descriptor has the form of a CHARACTER constant. It causes characters to be written from the enclosed characters (including blanks) of the edit descriptor itself.

The width of the field is the number of characters contained in, but not including, the delimiting apostrophes (quotation marks). Within the field, two consecutive apostrophes (quotation marks) are counted as a single apostrophe (quotation mark).

The statements:

```
      a = 123.45
      WRITE (*,41)a
41    FORMAT('0a=',F6.2)
```

print

```
a=123.45
```

The "0" in 0a= specifies carriage control.

The statements:

```
      a = 123.45
      b = 123.45
      c = 4.5
      WRITE(7,10)a,b,c
10    FORMAT ('a='F6.2,'b='F6.2,'c='F6.1)
```

cause

a=123.45b=123.45c=~~bbb~~4.5

to be written to file 7.

NOTE: If unit 7 is a formatted sequential file using FORTRAN Carriage Control, the leading "a" of the 'a=' in FORMAT statement 10 will be used for carriage control.

9.7.2 H Editing

The "nH" edit descriptor causes CHARACTER information to be written from the "n" characters (including blanks) following the "H" in the format specification itself.

Example:

```
          a = 123.45
          WRITE (*,10)a
    10    FORMAT(5H0a = , F6.2)
```

causes

```
    a = 123.45
```

to be printed. The "0" in 0a specifies carriage control.

9.7.3 Positional Editing

The "T" and "X" edit descriptors control what character position the next character is transmitted to or from in the current record. The position specified by a "T" edit descriptor may be in either direction from the current position.

An input record can be processed more than once, possibly with different editing. In sequential input, if a position beyond the last character of the record is specified, blank characters are transmitted from those positions. On output, a "T" edit descriptor may cause a character in the record to be replaced. On sequential output, if a position beyond the current length is specified, and any characters are then output, the positions not previously filled are filled with blanks. Example:

```
         a = 123.45
         b = 4.5
         WRITE(*,10) a,b
    10   FORMAT ('0a=',F6.2,5X,'b=',F3.1)
```

prints

a=123.45~~bbbbb~~b=4.5

9.7.4 T Editing

The "Tc" descriptor indicates that the transmission of the next character to or from a record is to occur at the "cth" character position relative to character one of the record.

The statements:

```
      READ(3,10)a
10    FORMAT(T5,F6.2)
```

cause REAL "a" to be read from the current record of file 3, starting in character position 5.

The "TLc" edit descriptor indicates that the transfer of the next character to or from the record is to occur at the character position "c" characters backward (to the left) from the current position.

The "TRc" edit descriptor indicates that the transfer of the next character to or from the record is to occur at the character position "c" characters forward (to the right) from the current position.

Example:

```
      a = 123.45
      b = 4.5
      WRITE (*,10) a,b
10    FORMAT ('0a=', F6.2, TL5, F3.1)
```

prints

```
a=14.545
```

9.7.5 X Editing

The "bX" edit descriptor indicates that the transfer of the next character to or from a record is to occur "b" characters to the right of the current position. The statements:

```
      a = 123.45
      b = 4.5
      WRITE (*,10) a,b
10    FORMAT ('0a=', F6.2, 3X, F3.1)
```

print

```
a=123.45   4.5
```

9.7.6 Slash Editing

The slash edit descriptor (/) causes the file to be positioned at the beginning of the next record, which then becomes the current record. An empty record may be created on output, and an entire record may be skipped on input. On output to a sequential file, a new record is created and becomes the last record in the file.

If, while reading a direct access file, the record number of the new record exceeds the number of records in the file, an end-of-file condition exists. The statements:

```
      READ (5,10)a,b,c
10    FORMAT (F6.2/F6.2/F6.2)
```

cause "a" to be read from the first record of unit 5, "b" from the second, and "c" from the third. A comma is optional preceding or following the slash edit descriptor. Example:

```
107    FORMAT(I5,/,I3)
```

and

```
107    FORMAT(I5/I3)
```

are equivalent.

9.7.7 Colon Editing

The colon edit descriptor (:) terminates format control if there are no more items in the input/output list. The colon edit descriptor has no effect if there are more items in the input/output list.

The statements:

```
      WRITE(5,10) 5.0
      WRITE(5,10) 5.0, 6.0
10    FORMAT(' five='F3.1:'six='F3.1)
```

cause

```
five=5.0
five=5.0six=6.0
```

to be written to the file designated by unit 5.

9.7.8 S Editing

The "S", "SP" and "SS" edit descriptors control whether an optional "+" is to be transmitted in numeric output fields. If an "SP" edit descriptor is encountered in a format specification, FORTRAN produces a plus sign in any subsequent position that normally would contain an optional plus.

If an "SS" or "S" edit descriptor is encountered, FORTRAN does not produce a plus sign in any subsequent position that normally contains an optional plus.

The "S", "SP" and "SS" edit descriptors affect "I", "F", "E", "D" and "G" editing only during an output statement. "S" edit descriptors have no effect on input statements or the exponent field in "E", "D" or "G" editing. The statements:

```
      a=12.34
      b=45.67
      c=67.89
      WRITE(UNIT=*,FMT=10)a,b,c
10    FORMAT('0',SP,F6.2,SS,F6.2,F6.2)
```

print

```
+12.34b45.67b67.89
```

The "0" is for carriage control.

9.7.9 P Editing

A scale factor is specified by a "P" edit descriptor, which is of the form:

```
kP
```

Where:

"k" is an optionally signed INTEGER constant called the scale factor.

The value of the scale factor is zero at the beginning of execution of each input/output statement. The edit descriptor "P" changes the scale factor to "k". That scale factor then applies to all subsequently interpreted "F", "E", "D" and "G" edit descriptors until another "P" edit descriptor changes it.

NOTE: Reversion of format control does not affect the established scale factor.

The scale factor "k" causes the following:

1. On input with the "F", "E", "D" and "G" edit descriptors in the field (and no exponent) and on output with "F" editing, the scale factor effect is that the externally represented number equals the internally represented number multiplied by 10**k. This means that on input, the number read is divided by 10**k, and on output, the output list item is multiplied by 10**k before being output. Where "testfile" contains 123.45, the statements:

```
   OPEN (1,'testfile')
   READ(UNIT=1,FMT=10) a
10 FORMAT(+3P,F8.4)
   PRINT *, a
```

print

.1234500

2. On input with "F", "E", "D" and "G" editing, the scale factor has no effect if there is an exponent in the input field.

3. On output with "E" and "D" editing, the basic REAL constant part of the quantity to be produced is multiplied by 10**k, and the exponent is reduced by k.

4. On output with "G" editing, the effect of the scale factor is suspended unless the magnitude of the datum to be edited is outside the range that permits the use of "F" editing. If the use of "E" editing is required, the scale factor has the same effect as with "E" output editing.

9.7.10 BN and BZ Editing

The "BN" and "BZ" edit descriptors may be used to specify the interpretation of nonleading blanks in numeric input fields.

The "BN" and "BZ" edit descriptors affect only "I", "F", "E", "D" and "G" editing during execution of an input statement. They have no effect during execution of an output statement. At the beginning of each formatted input statement, blank characters are interpreted as zeros or are ignored, depending on the value of the BLANK= specifier in the OPEN statement.

If a "BN" edit descriptor is encountered in a format specification, all blank characters in succeeding numeric input fields are ignored. However, a field of all blanks has the value of zero.

If a "BZ" edit descriptor is encountered in a format specification, all blank characters (other than leading blanks) in succeeding numeric fields are treated as zeros.

9.7.11 Numeric Editing

The "I", "F", "E", "D" and "G" edit descriptors specify input/output of numeric data. The following general rules apply:

1. On input, leading blanks are not significant. The interpretation of blanks other than leading blanks is determined by a combination of the value in the BLANK= specifier in the OPEN statement and the BN or BZ edit descriptor that is in effect (if any). Plus signs (+) may be omitted. A field of all blanks is considered to be zero.

2. On input with "F", "E", "D" and "G" editing, a decimal point appearing in the input field overrides the portion of an edit descriptor that specifies the decimal point location.

   A comma in a numeric input field terminates the field regardless of whether or not the specified width of the field has been exhausted.

   The input field may have more digits than FORTRAN uses to approximate the value of the datum. (For the exact number of decimal digits of precision, see **Appendix N**).

3. On output, the representation of a positive or zero value in the field may be prefixed with a plus sign (+), as controlled by the "S" edit descriptor. The representation of a negative value in a field is always prefixed with a minus sign (–).

4. On output, the representation is right-justified in the field. If the number of characters produced by the editing is less than the specified field width, leading blanks are inserted in the field.

5. On output, if the number of characters produced exceeds the field width, or if an exponent exceeds its specified length using the "Ew.dEe" or "Ew.dDe" edit descriptors, the entire field of width "w" is filled with

asterisks. If the "SP" edit descriptor is in effect, the plus sign (+) is not optional and causes overflow if there is not room for it in the field.

Example:

```
      a=123.45
      WRITE(UNIT=*,FMT=10)a
10    FORMAT(' ',F5.2)
```

prints

```
*****
```

6. Any of the numeric edit descriptors may be used with any numeric data type. Any necessary conversions will be performed automatically.

   A COMPLEX number requires two edit descriptors, the first for the real part and the second for the imaginary part. The two edit descriptors may be different.

9.7.12 I Editing

The "Iw" and "Iw.m" edit descriptors indicate that the external representation of the transmission list item is an integer and that the field to be edited occupies "w" positions. In the input field, the datum must be in the form of an optionally signed INTEGER constant, except for the interpretation of blanks. On input, an "Iw.m" edit descriptor is identical to an "Iw" edit descriptor.

On output, the field for the "Iw" edit descriptor consists of zero or more leading blanks followed by a minus sign (–) if the value of the datum is negative or a plus sign (+) if the "SP" edit descriptor is in effect, followed by the integer. An integer always consists of at least one digit.

On output, the field for the "Iw.m" edit descriptor is identical to the "Iw" edit descriptor, except that the integer is output with "m" minimum digits and, if necessary, is preceded by leading zeros to give the minimum length. The value of "m" must not be greater than "w". If "m" is zero and the value of the datum is zero, then the output field consists of only blank characters.

Example:

```
      j = 0
      k = 1
      m = -123
      n = 123
      WRITE (7,10)j,k,m,n
10    FORMAT(I2.0,I4.3,I5,SP,I5)
```

causes

```
ƀƀƀ001ƀ – 123ƀ + 123
```

to be written to file 7.

9.7.13 F Editing

The "Fw.d" edit descriptor indicates that the field occupies "w" positions; the fractional part consists of "d" digits. On input, the field consists of an optional plus or minus sign, followed by a sequence of digits which may contain a decimal point. This basic form may be followed by an exponent of one of the three following forms:

1. signed INTEGER constant.

   –123 + 2
   123 + 2
   +123 – 2

2. "E" followed by an optionally signed INTEGER constant.

   –123E2
   123E + 2
   123E – 2

3. "D" followed by an optionally signed INTEGER constant.

   –123D2
   123D + 2
   123D – 2

On output, the field consists of leading blanks (if necessary), followed by a minus sign if the value is negative or a plus sign if the "SP" edit descriptor is in effect, followed by a sequence of digits that contains a decimal point. The field is modified by the established scale factor and is rounded to "d" fractional digits.

9.7.14 D and E Editing

The "Ew.d", "Dw.d", "Ew.dEe", "Ew.dDe", "Dw.dEe" and "Dw.dDe" edit descriptors indicate that the external field occupies "w" positions, the fractional part of which consists of "d" digits, and the exponent part, of "e" digits.

On input, the field is the same as "F" editing; the exponent part of the edit descriptor is ignored.

On output, the field is adjusted to indicate the exponent of the datum. The field consists of leading blanks (if necessary), followed by a minus sign if the value is negative or a plus sign if the "SP" edit descriptor is in effect, followed by a zero (depending on scale factor), followed by a decimal point, followed by the "d" most significant digits of the datum after rounding off, followed by an "E" (or a "D", if it is specified as the exponent character in the edit descriptor), followed by a mandatory plus or minus sign (depending on the value of the exponent), followed by the exponent. The exponent is one of two forms:

1. For the "Ew.d" and "Dw.d" edit descriptors, the exponent is output as two digits.

2. For the "Ew.dEe", "Ew.dDe", "Dw.dEe" and "Dw.dDe" edit descriptors, the exponent is displayed in "e" digits.

Example:

```
      a = 123.45
      b = -123.45
      WRITE(7,10)a
      WRITE(7,20)b
10    FORMAT(E10.3)
20    FORMAT(E11.3E3)
```

causes

```
b0.123E + 03
-0.123E + 003
```

to be written to unit 7.

The scale factor "k" (set by the last "P" edit descriptor in the format specification) controls where the decimal point is placed. If "k" is zero (default), the decimal point is placed as shown above with one leading zero before the decimal point. If "k" is less than zero, then there is one leading zero followed by the decimal point followed by "k" zeros, followed by the value of the datum rounded to "d" – "k" digits. If "k" is greater than zero and less than "d" + 2, then there are exactly "k" significant digits to the left of the decimal point and "d" – "k" + 1 significant digits to the right of the decimal point. The value of "k" must lie within the range: "– d" < "k" < "d" + 2.

Example:

```
      a = 123.45
      WRITE (7,10)a
      WRITE (7,20)a
      WRITE (7,30)a
10    FORMAT(E11.5)
20    FORMAT(2P,E12.5)
30    FORMAT(-2P,E11.5)
```

causes

```
0.12345E + 03
12.34500E + 01
0.00123E + 05
```

to be written to unit 7. Note that the scale factor affects the value of the exponent that is printed and not how long the field is.

9.7.15 G Editing

The "Gw.d", "Gw.dEe" and "Gw.dDe" edit descriptors indicate that the field occupies "w" positions with "d" significant digits.

On input, "G" editing is the same as "F" editing.

On output, the method of representation in the output field depends on the magnitude of the internal datum. If the decimal point falls just before, within, or just after the "d" significant digits to be printed, then an appropriate "F" edit descriptor is used. If not, then the "E" edit descriptor is used. The scale factor "k", "Ee", and "De" edit descriptors have effect only if the magnitude of the datum causes the "E" edit code to be used. However, if the "F" edit descriptor is used, the quantity of blanks required for the exponent field (had the "E" edit descriptor been used) is still explicitly appended to the number output and included as part of the total width "w".

Example:

```
a = 7.0
b = .007
c = 700.0
d = .0000000007
WRITE (*,'(' 'G11.3)')a,b,c,d
```

prints

~~bbb~~7.00~~bbbb~~
~~bb~~0.700E – 02
~~bbb~~700.~~bbbb~~
~~bb~~0.700E – 09

9.7.16 L Editing

The "Lw" edit descriptor indicates that the field occupies "w" positions. The specified input/output list item must be LOGICAL.

On input, the list item is defined with a LOGICAL datum. The input field should consist of zero or more blanks, optionally followed by a period, followed by a "T" for true or "F" for false.

The "T" or "F" may be followed by additional characters in the field. On output, the field consists of "w" – 1 blanks followed by a "T" or "F" depending on the value of the datum. Example:

```
      LOGICAL a
      a = .TRUE.
      WRITE (7,10)a
10    FORMAT (L3)
```

prints

~~bb~~T

9.7.17 A Editing

The "A[w]" edit descriptor is used with CHARACTER input/output values. If a field width "w" is specified with an "A" edit descriptor, the field consists of "w" characters. Otherwise, the number of characters in the field is equal to the length of the CHARACTER transmission list item.

On input, if the specified field width "w" is greater than the length of the input character list item into which the characters are read, then the rightmost characters are used. If the specified field width "w" is less than the length of the input list item, then trailing blanks are added.

On output, if the specified field width "w" is less than the length of the output list item, then the leftmost "w" characters of the output list item are output. If the specified field width "w" is greater than the length of the output list item, then leading blanks are used.

NOTE: The "A" descriptor does not require the total character width. If not specified, the length of the item is used. For non-CHARACTER values the width of the field is type dependent (e.g., for INTEGER, 4; COMPLEX, 8).

Because any ASCII character is transmitted, it is possible to input or output control characters. Attempting to input or output ^Z (end- of-file), ^M (carriage return) or ^C (break) in a sequential file is not recommended and may produce undesirable results.

Example:

```
      CHARACTER a*4
      a = 'rstu'
      WRITE(*,'(' 'a4)') a
      WRITE(*,'(' ',a3)') a
      WRITE(*,30) a(1:3),a(2:4),a(1:3)
30    FORMAT(' ',3a4)
```

prints

rstu
rst
~~b~~rst~~b~~stu~~b~~rst

9.7.18 Z Editing

The "Zw" edit descriptor is used with non-CHARACTER input/output values. The list item is output as a hexadecimal value, right-justified in the field whose width is specified by "w". Leading zeros are added if necessary to complete the number of hexadecimal digits required to represent the number of bytes in the data type.

The statements:

```
REAL a
INTEGER i
INTEGER *2 j
i = 123456
j = 36
a = 123.45
PRINT '(Z10)', i, j, a
```

print

~~bb~~0001E240
~~bbbbbb~~0024
~~bb~~42F6E666

(The internal representation of i, j and a)

9.8 Namelist Input/Output

F77L's Namelist feature provides another method of performing formatted input and output. The namelist forms of READ, WRITE, INPUT and PRINT statements reference namelists instead of formats and input/output lists.

It is easiest to see how the namelist feature works in an example. Consider the following pair of statements:

```
NAMELIST/codes/i,j,x,y
.
.
.
WRITE(iofile,codes)
```

Assuming the program assigned appropriate values to the variables, the program would produce the following file record:

```
&CODES I=2001,J=30,X=1.5,Y=8.3333 &END
```

Certain characteristics of the namelist output record are notable:

The first field of the record is the namelist name prefixed with the character "&";

The output data is followed by the characters "&END";

INTEGER data are produced for INTEGER variables and REAL data are produced for REAL variables.

Commas separate output data fields. Blanks appear after the namelist name and before "&END".

With namelist-directed file input, the system searches the specified file sequentially until a record beginning with the namelist name, prefixed with an "&", is found.

Using the previous example record, the statements:

```
NAMELIST/codes/i,j,x,y
.
.
.
READ(iofile,codes)
```

cause the values 2001, 30, 1.5 and 8.3333 to be assigned to I, J, X and Y when the READ statement is executed.

To illustrate the use of an array name in a namelist, assume that appropriate values have been assigned to array "larry". In the program:

```
INTEGER larry(5)
NAMELIST/aname/larry
.
.
.
PRINT aname
```

the following line is displayed at the console:

```
&ANAME LARRY=922,1,308,26865,0 &END
```

The NAMELIST statement, a specification statement, is described in **Chapter 6**. How to use the NAMELIST statement to perform input/output is described in the following paragraphs.

9.8.1 Namelist Directed File Input/Output

The syntax for namelist-directed file READ and WRITE statements is:

```
READ ([spec=]sval,[spec=]sval[,spec=sval]...)

WRITE ([spec=]sval,[spec=]sval[,spec=sval]...)
```

Where:

"spec" is one of END, ERR, IOSTAT, NML, REC or UNIT;

"sval" is a numeric expression, CHARACTER expression or a statement label, depending on the "spec" used.

Two specifiers are required in the READ or WRITE statement: UNIT and NML. The other specifiers are optional.

Examples:

```
READ(5,NAMES)
WRITE(IOSTAT=530,UNIT=6,NML=OLIST)
```

The NML= Specifier

The NML= specifier is an input/output specifier used for namelist-directed file operations. The specifier both indicates that namelist-directed input/output is being used and specifies the namelist name. The specifier has the form:

```
[NML=]nlname
```

Where:

"nlname" is a namelist name in the current program unit.

The keyword NML= is optional if, and only if, the following conditions are met:

The NML= specifier is the second specifier in the statement; and

The first specifier in the statement is a UNIT= specifier in which the keyword UNIT= is omitted.

File Input

When a namelist-directed READ statement is executed, the file designated by the UNIT= specifier is searched for the record having the characters "&nlname" as the first nonblank characters. If the record is found, the input values are assigned to the indicated variables and arrays.

The input record has the following format:

```
&nlname [item=value[,[item=]value]...] &[END]
```

Where:

"nlname" is the namelist name; and

"item" is a variable, array name or array element name in the indicated NAMELIST statement;

"value" is an INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL or CHARACTER constant, as appropriate for "item". A LOGICAL constant is one of the following: T, .TRUE., F, .FALSE. or any value beginning with T or F.

The following rules apply to creating namelist input records:

The "nlname" cannot contain spaces and must be contained within a single record.

An item specified on the left side of the equal sign cannot contain spaces or tabs except within the parentheses of a subscript. Each item must be contained in a single record.

A null value is specified by two consecutive commas, by an initial comma or by a trailing comma. A null value indicates that the corresponding namelist array element is unchanged. A null value can represent an entire complex element but cannot be used for only one part of a complex constant.

The equal sign in a value assignment can be delimited by spaces.

Constants used in data items may take the following forms:

Constant Type	Examples
INTEGER	1, –868
REAL	1., –868.00
DOUBLE PRECISION	–86.8DI
COMPLEX	(1,–868.)
LOGICAL	T, .FALSE.
CHARACTER	'NOT YET', '1'

A symbolic constant (parameter) cannot be specified.

NOTE: A namelist input record does not have to contain values for each name listed in the NAMELIST statement. Also, a comma after the last value is optional.

File Output

When a namelist-directed WRITE statement is executed, the following elements are output to the designated file:

1. &nlnameƀ

2. the data fields, separated by commas

3. ƀ&END

Each variable and array name specified in the NAMELIST statement is output in a format appropriate for its type. The output field is large enough to contain all the significant digits. Each output field is labeled with its variable or array name.

9.8.2 Namelist Directed Console Input/Output

The INPUT, PRINT and short version of the READ statement provide namelist-directed console input/output.

The statement syntax is:

```
READ(*,[NML=]nlname)
READ nlname
INPUT nlname
PRINT nlname
```

Where:

"nlname" is a namelist name in the current program unit.

Console Input

For namelist directed input from the console, the following prompt is produced when the program is ready to accept input:

/nlname/?

Where:

"nlname" is a namelist name in the current program unit.

At this point data can be provided for any or all of the variables and arrays listed in the corresponding NAMELIST statement, in any order.

The response has the following format:

item=value[<sep>item=value]...

Where:

"item" is a variable, array name or array element name in the corresponding NAMELIST statement;

"value" is a constant of the type that is appropriate for "item"; and

"<sep>" is a comma, tab or space.

Follow the last value with:

~~b~~&[end]

to terminate the list of data.

Enter a carriage return to continue entry on the next line. With multiple line input, terminate the last entry on the last line with:

~~b~~&[end]

Example:

Using the introductory example again, the following lines could be used to assign values to the listed variables:

```
/CODES/?  x = 0,y = 3.141592
/CODES/?  j = 80061 &
```

Console Output

With namelist directed output to the console, the output data is produced in the same format shown for file output. That is, data items are separated by commas. The record begins with "&nlnameƀ" and ends with "ƀ&END".

9.8.3 Assigning Consecutive Values

A single value can be assigned to consecutive array elements by the specification of a repeat count with the constant, as follows:

```
item=r*value
```

Where:

"item" is an unsubscripted array name in a NAMELIST statement;

"r" is a nonzero, unsigned INTEGER constant; and

"value" is a constant of a type that is appropriate for "item".

The form

```
r*
```

can be used to assign null values (that is, to leave array elements unchanged).

Example:

Consider the following program:

```
REAL ax(4,5)
NAMELIST/anames/ax
.
.
.
READ(infile,anames)
```

If the file "infile" contains the following record:

```
&ANAMES AX=4*2.222 &END
```

the execution of the READ statement would cause the following assignment of values to take place:

```
AX(1,1) = 2.222
AX(2,1) = 2.222
AX(3,1) = 2.222
AX(4,1) = 2.222
```

9.8.4 Forms of Input Data Items

With both console and file input, the general form of an input data item:

```
item=value
```

can take three specific forms, as described below.

<u>Variable Name = Constant</u>

The variable name may be an array element name or a variable name. Subscripts must be INTEGER constants.

Examples:

```
icon = 12345
harry(3) = 1.2345E + 4
```

Array Name = Set of Constants (separated by commas)

The number of constants in the set must be less than or equal to the number of elements in the array. A repeat count, as shown above, may be used to assign consecutive values in the array.

Examples:

Assume arrays LARRY and HARRY have five elements each:

```
larry=922,1,308,26865,0
harry=0,2*,1.2345E + 4
```

In the second line, values are assigned to elements one and four of HARRY. Elements two, three and five are not changed.

Subscripted Array Name = Set of Constants (separated by commas)

The set of constants is assigned to the array, starting with the specified element. Unspecified array elements are not altered.

Examples:

Assume arrays LARRY and HARRY have five elements each.

```
larry(1)=922,1,308,26865,0
harry(4)=2*1.2345E + 4
```

All five elements of LARRY and elements four and five of HARRY are assigned values with these input data items.

NOTE: Namelist output is not always proper namelist input when using CHARACTER data. For example, a CHARACTER value using embedded commas, blanks etc. will result in those characters being incorrectly interpreted as delimiters upon input. In general, namelist output parallels list-directed output where quotation mark delimiters do not appear on output.

9.9 CLOSE Statement

The CLOSE statement is used to terminate the connection of a file to a unit. All open files are automatically closed upon execution of a STOP (or an END statement in a main program unit). The syntax for the statement is:

```
CLOSE ([spec=]sval [,spec=sval]...)
```

Where:

"spec" is one of UNIT, IOSTAT, ERR or STATUS, in any order; and

"sval" is a numeric or CHARACTER expression or a statement label.

UNIT=, IOSTAT= and ERR= are described in Section 9.1. The remaining specifier is used as follows.

STATUS= CHARACTER Expression

The CHARACTER expression determines whether the file being closed is to be kept by the system for later use or to be deleted from the user's directory. The possible options are "KEEP" and "DELETE". The default is "KEEP" for nonscratch files.

The statement:

```
50    CLOSE(UNIT=1,ERR=100,STATUS='KEEP')
```

is equivalent to

```
50    CLOSE(1,ERR=100)
```

because "KEEP" is the default and "UNIT=" is optional if the first specifier in the control list is the numeric expression specifying the unit number.

9.10 INQUIRE Statement

The INQUIRE statement is used to determine the properties of a file. Either a unit number or filename may be used to identify the file. The file does not have to be connected to the specified unit number; if it is connected, it need not have been connected in the current program unit.

The INQUIRE statement returns information regarding the following file properties: existence, opened, unit number, named, name, access type, format type, record length, treatment of blanks and next record available. The syntax is:

```
INQUIRE (spec=sval [,spec=sval]...)
```

Where:

"spec" is one of: UNIT, FILE, ERR, IOSTAT, OPENED, NUMBER, NAMED, NAME, ACCESS, FORM, SEQUENTIAL, DIRECT, FORMATTED, UNFORMATTED, RECL, BLANK, EXIST, NEXTREC,

FLEN, POSITION, ACTION, DELIM or PAD

in any order; and

"sval" is a statement label or LOGICAL, INTEGER, CHARACTER variable or array element, depending on the "spec" used.

Much of the information sought by an INQUIRE statement is described in Section 9.3, **OPEN Statement**. Either the FILE= or UNIT= specifier must be present to inquire about a given filename or unit number, but not both. The ERR= and IOSTAT= specifiers are used as described in Section 9.1. Other specifiers used by INQUIRE are described below.

OPENED= LOGICAL Variable or Array Element

The LOGICAL target is assigned ".TRUE." if the specified file is connected to a unit or the specified unit is connected to a file and ".FALSE." if there is no connection.

NUMBER= INTEGER Variable or Array Element

The INTEGER target is assigned the value of the unit number of the connected file. The value is undefined if the file is not connected.

NAMED= LOGICAL Variable or Array Element

The LOGICAL target is assigned the value ".TRUE." if the file has a name, and ".FALSE.", otherwise.

NAME= CHARACTER Variable or Array Element

The target is assigned the name of the file.

ACCESS= CHARACTER Variable or Array Element

The target is assigned the value corresponding to the specification in the OPEN statement, such as "DIRECT" or "SEQUENTIAL". See the discussion of ACCESS= in Section 9.3 for other possible values.

FORM= CHARACTER Variable or Array Element

The target is assigned the value "FORMATTED" or "UNFORMATTED", depending on the FORM= specifier in the OPEN statement.

SEQUENTIAL= CHARACTER Variable or Array Element

The target is assigned the value "YES" if SEQUENTIAL is an allowed access method for the file, "NO" if SEQUENTIAL is not allowed, or "UNKNOWN" if the processor is unable to determine whether sequential access is permitted.

DIRECT= CHARACTER Variable or Array Element

The target is assigned the value "YES" if DIRECT is an allowed access method for the file, "NO" if DIRECT is not allowed or "UNKNOWN" if the processor is unable to determine whether DIRECT access is permitted.

FORMATTED= CHARACTER Variable or Array Element

The target is assigned the value "YES" if FORMATTED is an allowed form for the file, "NO" if FORMATTED is not an allowed form and "UNKNOWN" if the processor is unable to determine whether the file is formatted or unformatted.

NEXTREC= INTEGER Variable or Array Element

The INTEGER target is assigned the value "n" + 1, where "n" is the record number of the last record read or written on the file. If the file is connected but no records have been read or written since the connection, the value 1 is assigned. If the position of the file is indeterminate because of a previous error condition, the value assigned is undefined.

UNFORMATTED=CHARACTER Variable or Array Element

The target is assigned the value "YES" if UNFORMATTED is an allowed form for the file, "NO" if UNFORMATTED is not an allowed form, and "UNKNOWN" if the processor is unable to determine whether the file is formatted or unformatted.

RECL= INTEGER Variable or Array Element

The INTEGER target is assigned the value of the record length specified by the RECL= specifier in the OPEN statement.

BLANK= CHARACTER Variable or Array Element

The designated variable or array element is assigned the value "NULL" if null blank control is in effect for the file and is assigned the value "ZERO" if zero blank control is in effect. The variable or array element becomes undefined if the file does not exist or is not formatted.

EXIST= LOGICAL Variable or Array Element

The LOGICAL target is assigned ".TRUE." if the file referenced by the FILE= specifier exists and ".FALSE.", otherwise.

FLEN= INTEGER Variable or Array Element

The INTEGER target is assigned the length of the file (the number of bytes in the file).

POSITION= CHARACTER Variable or Array Element

There are four options for this specifier: "REWIND", "APPEND", "ASIS" and "UNDEFINED". The "REWIND" specifier is used if the file is connected by an OPEN statement for positioning at its initial point. The "APPEND" specifier is used if the file is connected for positioning at its terminal point. The "ASIS" specifier is used if the file is connected without changing its position. If this specifier is omitted, the default value is "UNDEFINED". If the file has been repositioned since the connection, the value "UNDEFINED" is assigned.

ACTION= CHARACTER Expression

There are three possible options for this specifier: "READ", "WRITE" and "READ/WRITE". "READ" specifies that the "WRITE". "PRINT" and "ENDFILE" statements must not refer to this connection. "WRITE" specifies that "READ" statements must not refer to this connection. "READ/WRITE" permits any I/O statements to refer to this connection. If this specifier is omitted, the default value is "READ/WRITE".

DELIM= CHARACTER Variable or Array Element

There are four options for this specifier: "APOSTROPHE", "QUOTE", "NONE" and "UNDEFINED". The specifier is assigned the value "APOSTROPHE" if the apostrophe is to be used to delimit CHARACTER data written by list-directed or namelist formatting. If the quotation mark is used to delimit these types of data, the specifier is assigned the value "QUOTE". If neither apostrophe or quotation mark are used to delimit CHARACTER data, the specifier is assigned the value "NONE". If there is no connection or if the connection is not for formatted I/O, the specifier is assigned the value "UNDEFINED".

PAD= CHARACTER Variable or Array Element

There are two options for this specifier: "YES" and "NO". The specifier is assigned the value "NO" if the connection of the file to the unit included the PAD= specifier and its value was "NO". Otherwise, the specifier is assigned the value "YES".

9.11 REWIND Statement

The REWIND statement is used to reposition a file at the beginning of its first record. The syntax is:

```
REWIND unitnum
REWIND (spec=sval [,spec=sval]...)
```

Where:

"unitnum" is a positive numeric expression;

"spec" is either UNIT, IOSTAT or ERR, in any order; and

"sval" is a positive numeric expression (UNIT), a statement label (ERR), or an INTEGER variable or array element (IOSTAT).

The UNIT=, IOSTAT= and ERR= specifiers are used as described in Section 9.1.

Examples:

```
REWIND n
REWIND(UNIT=1,ERR=100)
REWIND(ERR=100,UNIT=1)
REWIND(1)
```

9.12 BACKSPACE Statement

Execution of a BACKSPACE statement causes the file connected to the specified unit to be positioned before the preceding record. If the file is at its initial point, then the BACKSPACE statement has no effect. The syntax is:

```
BACKSPACE unitnum
BACKSPACE (spec=sval [,spec=sval]...)
```

Where:

"unitnum" is a positive numeric expression;

"spec" is one of UNIT, ERR or IOSTAT, in any order; and

"sval" is a positive numeric expression (UNIT), a statement label (ERR), or an INTEGER variable or array element (IOSTAT).

The UNIT=, ERR= and IOSTAT= specifiers are used as described in Section 9.1.

Examples:

```
BACKSPACE infil
BACKSPACE (6, IOSTAT=IERR)
```

9.13 ENDFILE Statement

The ENDFILE statement is used to truncate a file at its current position. This causes records that are past this point to be lost. For sequential files an end-of-file record is written, but it contains no user data. The syntax is:

```
ENDFILE unitnum
ENDFILE (spec=sval [,spec=sval]...)
```

Where:

"unitnum" is a positive numeric expression;

"spec" is UNIT, ERR or IOSTAT, in any order; and

"sval" is a positive numeric expression (UNIT), a statement label (ERR), or an INTEGER variable or array element (IOSTAT).

The UNIT=, ERR= and IOSTAT= specifiers are used as described in Section 9.1.

Examples:

```
ENDFILE(UNIT=1,ERR=10)
ENDFILE(2,ERR=10)
ENDFILE 5
```

9.14 IOSTAT Error Codes

Execution of an input/output statement in which the IOSTAT= specifier is present causes FORTRAN to return a value to the INTEGER variable or array element target of the IOSTAT= specifier. If an error occurs, the execution sequence is affected as follows:

If neither IOSTAT= nor ERR= specifiers are present, an error message is printed and execution is halted.

If the IOSTAT= specifier is present without an ERR= specifier, execution continues sequentially after the IOSTAT target is defined.

If both IOSTAT= and ERR= specifiers are present, execution continues where designated by the ERR= label after the IOSTAT target is defined.

The value returned to the IOSTAT target depends upon the result of the input/output operation. The following general rules apply:

if an end-of-file condition occurs, the value returned is negative.

if an error condition occurs, the value returned is positive.

if neither an end-of-file nor an error condition occurs, the value returned is zero.

The following procedure may be used to convert the IOSTAT error code to the equivalent value listed in **Appendix L**.

```
        OPEN (1,'file1',IOSTAT=errcode,ERR=500)
        ...
500     PRINT, 'error code =',MOD(errcode, 256)
        ...
```

The value returned by the IOSTAT= specifier is a 16-bit machine word, divided into the two fields:

Bits	Contents
0-7	Error message printing flags
8-15	Error code

The data in bits 0 through 7 are used by the internal FORTRAN runtime routines.

EXECUTABLE PROGRAMS 10

EXECUTABLE PROGRAMS 10

10 EXECUTABLE PROGRAMS

An executable program is a collection of program units consisting of exactly one main program and any number of subprograms. Execution of a FORTRAN program begins with the first executable statement of the main program.

A program unit consists of a sequence of statements and optional comment lines. A program unit is either a main program or a subprogram. Every program unit must physically terminate with an END statement. A main program may begin with a PROGRAM statement although it is not required. Other program units begin with a FUNCTION, SUBROUTINE or BLOCK DATA statement. The different kinds of program units and their related statements are described in this chapter.

A procedure is a sequence of statements that performs one or more tasks. Each program unit, other than block data subprograms, is a procedure. Each subprogram is an external procedure to the main program. Internal procedures are procedures which are defined within program units. Single-line functions are the only internal procedures allowed in FORTRAN 77.

Functions and subroutines are two kinds of external procedures. A function returns a single value. Functions may be single-line statements within procedures or they may be external procedures.

The functions used in FORTRAN (described in Section 10.2 below and **Chapter 11**) may be categorized as follows:

intrinsic functions required by the ANSI Standard (e.g., SIN).

Intrinsic functions can also be those not required by the standard but provided in F77L to make better use of the features of the computer and solve common problems (e.g., CHARNB)

statement single-line, user-defined functions

external subprogram, user-defined functions

In FORTRAN, subroutines are always external procedures and are referenced by CALL statements. A subroutine performs some action or task for another program unit. The structure and use of subroutines are described in Section 10.3.

Subprograms are either functions, subroutines or a third type of subprogram called a block data subprogram. Block data subprograms are used to provide initial values for variables and array elements allocated in named common blocks (see Section 6.4). Block data subprograms are described in Section 10.6.

10.1 Main Programs

A main program is the program unit that begins each executable program. There can be only one main program unit for an executable program. A main program may begin with a PROGRAM statement although it is not required. More particularly, a main program does not begin with a FUNCTION, SUBROUTINE or BLOCK DATA statement.

A main program may contain any statement except BLOCK DATA, FUNCTION, SUBROUTINE, ENTRY, RETURN or any additional PROGRAM statements. A main program may not be referenced from a subprogram or from itself.

10.1.1 PROGRAM Statement

A PROGRAM statement is not required to appear in an executable program. If it does appear, it must be the first statement of the main program. The format of the statement is:

```
PROGRAM pname
```

Where:

"pname" is the symbolic name of the main program in which the PROGRAM statement appears.

The symbolic name is global to the executable program and must not be the same as the name of an external procedure or block data subprogram in the same executable program. The name must not be the same as any local name in the main program. Names must begin with a letter and may be up to 31 characters in length (see Section 1.4.5).

10.2 Functions

Two kinds of functions: statement and external, are described in this section. Intrinsic functions are described and listed in **Chapter 11**. Statement and external functions are supplied by the programmer. Statement functions are single-line functions, external functions are defined in separate subprograms.

A function is referenced in an expression and supplies a value to the expression. The value supplied is the value of the function. The syntax for function reference is:

```
fun([a[,a]...])
```

Where:

"fun" is the symbolic name of a function; and

"a" is an actual argument.

A function reference may appear only as an operand in a numeric, LOGICAL or CHARACTER expression (see **Chapter 4**). Execution of a function reference in an expression causes the evaluation of the function identified by "fun". When a reference appears in an argument list of a CALL statement, the function is evaluated and the resulting value is returned. The data type of the result of a statement function or external function reference is the same as the type of the function name. The data type of the value of an intrinsic function is specified in **Chapter 11**.

A statement function may be referenced only in the program unit in which it appears. A function subprogram may be referenced in any program unit of an executable program.

Statement Functions

A statement function is a procedure specified by a single statement in a program unit and has syntax similar to an assignment statement. The syntax for definition of a statement function is:

fun ([d[,d]...]) = e

Where:

"fun" is the symbolic name of the statement function;

"d" is a statement function dummy argument; and

"e" is an expression where each operand of "e" must be one of the following:

1. a constant;
2. a symbolic name of a constant;
3. a variable reference;
4. an array element reference;
5. an intrinsic function reference;

6. a reference to another statement function which has already been defined in the program unit;

7. an external function reference;

8. a dummy procedure reference; or

9. an expression enclosed in parentheses.

A statement function itself is not executable. It must appear after the specification statements and before the first executable statement of the program unit in which it is used. A statement function can be used only in the program unit in which it is defined.

The relationship between "fun" and "e" conforms to the same rules as assignment statements (see Sections 7.1 and 7.2).

The type of a statement function is determined by the type of its name (see Section 2.1).

The symbolic name of a statement function is a local name and must not be the same as the name of any other entity in the program unit, other than the name of a common block.

Here is an example of the definition and use of a statement function, "square":

```
PROGRAM demo
square (x) = x*x
a = 4.0
a = square (a)
PRINT *, a
END
```

This program prints the following result:

~~bbb~~16.00000

External Functions and Function Subprograms

An external function is a referenced function that is not part of the program unit that references it. An external function is either a FORTRAN function subprogram or a function written in another language. An ENTRY statement may be used within an external function to allow reference to some point other than the first executable statement (see Section 10.4).

A function subprogram is a program unit that specifies one or more external functions. A function subprogram has a FUNCTION statement as its first statement.

10.2.1 FUNCTION Statement

A FUNCTION statement must appear at the beginning of a function subprogram. The statement indicates the subprogram name and any dummy argument names. A type specification may also be included. The syntax for the statement is:

```
[RECURSIVE][typ[*len]] FUNCTION fun([d[,d]...])

      or

[typ[*len]][RECURSIVE] FUNCTION fun([d[,d]...])
```

Where:

"RECURSIVE" is a Fortran 8x keyword which may be included before the FUNCTION statement;

"typ" is one of LOGICAL, CHARACTER, INTEGER, REAL, DOUBLE PRECISION or COMPLEX;

"*len" is the type length specification (see Section 2.1) of the function result;

"fun" is the symbolic name of the function subprogram in which the FUNCTION statement appears; and

"d" is a dummy argument; it is a variable name, array name or procedure name.

Within the function subprogram, the subprogram name "fun" must be assigned a value. "fun" may be assigned a value in an assignment statement, as a DO variable in the list of an input statement or as an argument for a subprogram reference in the same program unit. The last value assigned to "fun" during the execution of the function subprogram is the value that is returned by that function. If a data type is not specified for the function name, then the data type is determined by the implicit type rules.

Example:

```
PROGRAM main
PRINT *, square (4.)
END
REAL FUNCTION square (x)
square = x*x
RETURN
END
```

prints

~~bbb~~16.00000

A function subprogram may also assign a value to one of its dummy arguments, "d". If this happens, then the value of the corresponding actual argument is altered accordingly.

The RETURN statement returns control to the calling subprogram; the execution of the END statement has the same effect. The RETURN statement is described in Section 10.5.

<u>Function Subprogram Arguments</u>

The actual arguments in a function-subprogram reference in a program unit must agree in order, number and type with the corresponding dummy arguments in the FUNCTION statement of the referenced function.

An actual argument in a function subprogram reference can be one of the following:

1. an expression;
2. an array name;
3. an intrinsic function name; or
4. an external procedure name.

If an actual argument in a function-subprogram reference is of type CHARACTER, then its length must be greater than or equal to the length of its corresponding dummy argument in the FUNCTION statement of the referenced function. If the dummy argument length "len" is less than the actual argument length, the leftmost characters of the actual argument are used. When the dummy argument of type CHARACTER is an array, the length of the entire array, not of each array element, must be less than or equal to a corresponding actual argument length. If the dummy argument, whether array or variable, is defined with a length of (*), the dummy argument length is set to the actual argument length at entry time.

Example:

```
PROGRAM main
CHARACTER*10 a,b,c,d
...
result = mvch(a,b,c,d)
...
END
FUNCTION mvch(w,x,y,z)
CHARACTER w*3(3),x*10,y*4(3),z*(*)
...
```

"y" is illegal because its size (12 bytes) is greater than the actual argument's size (10 bytes). On the other hand, "z" would be legal no matter what size the actual argument is.

If the reference to a function subprogram is made within another function subprogram, it is possible to pass an argument through several function subprograms.

The program:

```
PROGRAM main
PRINT *, square (4.0)
END
REAL FUNCTION square (x)
REAL multiply
square = multiply (x,x)
END
REAL FUNCTION multiply (y,z)
multiply = y*z
END
```

prints

```
ƀƀƀ16.00000
```

Function Usage

A function subprogram may contain any statement except a BLOCK DATA, SUBROUTINE, PROGRAM or another FUNCTION statement. The symbolic name of a function subprogram is a global name and must not be the same as any other global name in that executable program. If the type of a function is specified in the FUNCTION statement, the function name must not appear in a contradictory type statement.

A function subprogram may reference itself, either directly or indirectly. For example, the following program is legal, recursively, because "square" references "multiply", which, in turn, references "square". However, it would abort because there is no end to the recursive loop.

```
PROGRAM main
PRINT *, square (b)
END
RECURSIVE REAL FUNCTION square (x)
REAL multiply
square = multiply (x)
END
REAL RECURSIVE FUNCTION multiply (x)
multiply = square (x)
END
```

In a function subprogram, the symbolic name of an argument in the FUNCTION statement is local to that program unit and must not appear in an EQUIVALENCE, PARAMETER, SAVE, INTRINSIC, DATA or COMMON statement. The name can be a common block name.

Here is a final example to illustrate how to pass values in a named common instead of as arguments:

```
PROGRAM main
COMMON/table/a,b,c
a = 2.0
b = 4.0
c = 6.0
d = add()
PRINT *, d
END
REAL FUNCTION add
COMMON/table/x,y,z
add = x + y + z
END
```

This program prints:

~~bbb~~12.00000

10.3 Subroutines

A subroutine subprogram is a program unit that has a SUBROUTINE statement as its first statement. It is used to perform an action or subtask for another program unit when it is referenced by a CALL statement. An ENTRY statement may be used within a subroutine to allow reference to some point other than the first executable statement. The ENTRY statement is described in Section 10.4. Subroutines may also include RETURN statements, which can specify that control pass to some point other than the next executable statement following the subroutine reference (see Section 10.5).

An argument list may be used to pass values to and from a subroutine. One or more of the dummy arguments of a subroutine subprogram may become defined or redefined during its execution and, thus, returned as resulting values to the actual arguments of the program unit referencing the subroutine.

F77L provides some user-callable subroutines, described in **Chapter 12**

10.3.1 SUBROUTINE Statement

A SUBROUTINE statement begins a subroutine subprogram, indicating the subroutine name and any dummy argument names. The syntax for the statement is:

[RECURSIVE] SUBROUTINE sub [([d[,d]...])]

Where:

"RECURSIVE" is a Fortran 8x keyword which may be included before the SUBROUTINE statement;

"sub" is the symbolic name of the subroutine subprogram in which the SUBROUTINE statement appears; and

"d" is an argument to the subroutine. "d" can be a variable name, array name, procedure name or an asterisk (indicating an alternate return specifier; see below).

A subroutine name is referenced in a CALL statement (described below). The actual arguments in the reference to a subroutine must agree in order, number and type with the corresponding dummy arguments in the SUBROUTINE statement of the called subroutine subprogram.

A subroutine name has neither type nor value. A name must begin with a letter and may be up to 31 characters in length (see Section 1.4.5). To pass a value between a calling program unit and a subprogram, the arguments or variables in a COMMON statement can be used.

10.3.2 CALL Statement

A subroutine is referenced by a CALL statement that invokes the subroutine subprogram using the name specified, passing actual arguments to replace the dummy arguments in the SUBROUTINE definition. The syntax is:

CALL sub [([a[,a]...])]

Where:

"sub" is either the name of a subroutine or a dummy procedure name that has been passed to the program unit containing the CALL statement; and

"a" is either an actual argument for the subroutine, which follows the same rules as arguments for function subprograms (see Section 10.2) or an <u>alternate return specifier</u> of the form "*s", where "s" is the statement label of an executable statement in the same program unit as the CALL statement.

Execution in a subroutine begins with the first executable statement after the SUBROUTINE or ENTRY statement referenced by the name in the CALL statement. Control returns to the next executable statement after the CALL statement, unless an alternate return (see RETURN statement, below) has been executed in the invoked subroutine.

<u>Subroutine Usage</u>

A subroutine may be referenced by any other program unit of the executable program, as well as by itself (i.e., subprograms may be recursive).

A subroutine subprogram may contain any statement except a BLOCK DATA, FUNCTION, PROGRAM or another SUBROUTINE statement.

A subroutine name is a global name and must not be the same as any other global name in the executable program or any local name in the subroutine program unit.

A dummy name in a subroutine is local to that subroutine and must not appear in an EQUIVALENCE, PARAMETER, SAVE, INTRINSIC, DATA or COMMON statement except as a named common block name.

Example:

```
PROGRAM main
EXTERNAL multiply
b = 4.0
CALL square(multiply,b)
PRINT *,b
END

SUBROUTINE square(dummy,x)
CALL dummy(x)
RETURN
END

SUBROUTINE multiply(z)
z = z*z
RETURN
END
```

prints

```
bbb16.00000
```

In this example, the name of a subroutine "multiply" is passed as an argument, so that subroutine "square" could use it to call another subroutine.

10.4 ENTRY Statement

The ENTRY statement allows entry to subroutine or function subprograms at points other than the first executable statement following the SUBROUTINE or FUNCTION statement. The entry point is defined by a name and may use dummy arguments. A subroutine or function may have one or more ENTRY statements.

The syntax for the statement is:

```
ENTRY en[([d[,d]...])]
```

Where:

"en" is the name of an entry in a function or subroutine and is called an entry name; and

"d" is an argument. It can be a variable name, array name or dummy procedure name. If the ENTRY statement appears in a subroutine then "d" can also be an asterisk, indicating an alternate return (see RETURN statement, below).

An ENTRY statement is not executable. It may appear anywhere within a function or subroutine after the FUNCTION or SUBROUTINE statement, except that an ENTRY statement must not appear within the range of a DO loop or within an IF...THEN...ELSE structure.

When an ENTRY statement is the entry point to a subprogram, execution begins with the first executable statement after the ENTRY statement.

An entry name is global and must not name another FUNCTION, SUBROUTINE or ENTRY.

If an ENTRY statement is in a subroutine, the entry name is referenced in a CALL statement. If the ENTRY statement is in a function subprogram, the entry name is used like a function name. If the ENTRY statement is in a function subprogram, then the entry name may appear in a type statement.

The order, number, type and names of the dummy arguments in an ENTRY statement may be different from the order, number, type and names of the dummy arguments in the FUNCTION, SUBROUTINE or other ENTRY statement in the same subprogram. However, each reference to a function, subroutine or entry point must use an actual argument list that agrees in order, number and type with the dummy argument list in the corresponding FUNCTION, SUBROUTINE or ENTRY statement.

Example:

```
PROGRAM main
a = 2.0
CALL square(a)
b = 2.0
CALL quad(b)
PRINT *,a,b
END

SUBROUTINE quad(x)
x = x*x
ENTRY square(x)
x = x*x
RETURN
END
```

prints

```
ƀƀƀ4.000000ƀƀƀƀƀƀƀ16.00000ƀƀƀ
```

10.5 RETURN Statement

A RETURN statement causes a return of control to the invoking program unit. A RETURN may appear only in a function or subroutine subprogram. In subroutines, the RETURN statement may specify an alternate return.

The syntax is:

RETURN [e]

Where:

"e" is a numeric expression that can appear only in a subroutine return and specifies an alternate return as described below.

Execution of the RETURN statement terminates execution of the function or subroutine in which it appears and returns control to the program unit that referenced the subprogram.

If the subprogram is a subroutine, control transfers to the next executable statement following the subroutine reference unless an alternate return is selected. If the subprogram is a function, control transfers to the next step of the process being performed by the statement that contains the function reference.

A subroutine or function does not require a RETURN statement. The execution of an END statement in a subprogram has the same effect as the execution of a RETURN statement.

A subprogram may contain more than one RETURN statement; however, they each have the same effect when executed. For example, the following subroutines: "aaa", "bbb" and "ccc", are equivalent:

```
      SUBROUTINE aaa(a)
      IF (a.LT.1) GO TO 10
10    RETURN
      END

      SUBROUTINE bbb(a)
      IF (a.LT.1) RETURN
      RETURN
      END

      SUBROUTINE ccc(a)
      IF (a.LT.1) GO TO 10
10    END
```

Alternate Returns from a Subroutine

Normally, control is returned from a subroutine to the program that calls it at the next executable statement after the CALL statement. By using the alternate form of the RETURN statement, control may be returned to any of the labeled executable statements in the calling program unit.

Use of the alternate form of the RETURN statement requires the use of the alternate return specifiers in the argument list of the CALL statement referencing this subroutine and the use of corresponding asterisks in the SUBROUTINE or ENTRY statements of this subroutine.

The alternate return specifiers in the CALL statement must have a one-to-one correspondence to the asterisks in the corresponding ENTRY or SUBROUTINE statement. For example, if there are three alternate return specifiers in a CALL statement, there must be three asterisks in the same positions in the corresponding ENTRY or SUBROUTINE statement.

Example:

```
CALL AAA(*10,*20,X,*30)
...
SUBROUTINE AAA(*,*,Y,*)
...
RETURN e
```

If the expression "e" in an alternate return statement evaluates to less than one or greater than the number of alternate return specifiers in the CALL statement, and that RETURN statement is executed, control is returned to the first executable statement after the CALL statement, as if it were a normal return.

The alternate return statements correspond to the alternate return specifiers such that:

RETURN 1	Refers to the first alternate return specifier in the argument list.
RETURN 2	Refers to the second specifier in the argument list.
...	
RETURN n	Refers to the "nth" specifier in the argument list.

Examples:

```
      READ *,i
      CALL bbb(i,*20,*30,*40)
20    a = 4.0
      GO TO 50
30    a = 6.0
      GO TO 50
40    a = 8.0
50    PRINT *,a
      END

      SUBROUTINE bbb(k,*,*,*)
      IF (k)10,20,30
10    RETURN 1
20    RETURN 2
30    RETURN 3
      END
```

After execution, if "i" is less than zero, then "a" equals 4.0; if "i" equals zero, then "a" equals 6.0; and if "i" is greater than zero, then "a" equals 8.0.

```
      READ *,i
      CALL bbb(*20,i,*30,*40)
20    a = 4.0
      GO TO 50
30    a = 6.0
      GO TO 50
40    a = 8.0
50    PRINT *,a
      END

      SUBROUTINE bbb(*,j,*,*)
      RETURN j
      END
```

After execution, if "i" equals 1, then "a" equals 4.0; if "i" equals 2, then "a" equals 6.0; if "i" equals 3, then "a" equals 8.0; otherwise, "a" equals 4.0.

10.6 Block Data Subprograms

Block data subprograms are used to provide initial values for variables and arrays allocated in named common blocks (see Section 6.4). A block data subprogram is a program unit that has a BLOCK DATA statement as its first statement. The subprogram is not executable. There may be more than one block data subprogram in an executable program.

10.6.1 BLOCK DATA Statement

A BLOCK DATA statement specifies that the subprogram that follows is a block data subprogram. The syntax is:

```
BLOCK DATA [bname]
```

Where:

> "bname" is the symbolic name of the block data subprogram in which the BLOCK DATA statement appears.

The optional "bname" is a global name and must not be the same as the name of an external procedure, main program or other block data subprogram in the same executable program.

The name must not be the same as any local name in that block data subprogram. Names must begin with a letter and may be from 1 to 31 characters in length (see Section 1.4.5).

NOTE: If a block data subprogram unit is to be loaded from a library, it must have a name which must be referenced by a loaded program unit by appearing in an EXTERNAL statement.

The BLOCK DATA statement must appear only as the first statement of a block data subprogram. The only other statements that may appear in a block data subprogram are IMPLICIT, SAVE, PARAMETER, DIMENSION, COMMON, EQUIVALENCE, DATA, END, INCLUDE and the type statements.

Only entities in a named common block may be initialized in a block data subprogram.

If an entity in a named common block is initialized, then all other entities in that common block must be specified even if they are not initialized.

More than one named common block may have entities initialized in a single block data subprogram.

Example:

```
BLOCK DATA initials
COMMON/table/a,b,c
COMMON/list/d(10,10)
DATA a/2.0/
DATA d/10*10*2.0/
END
```

INTRINSIC FUNCTIONS 11

SYSTEM SUBROUTINES 12

INTRINSIC FUNCTIONS 11

SYSTEM SUBROUTINES 12

11 INTRINSIC FUNCTIONS

Intrinsic functions are built into the FORTRAN Language. Some Intrinsic functions have generic names that identify their general purposes, such as SQRT for square root, and intrinsic names that identify their exact purposes in relation to its data type, such as DSQRT for the square root of a DOUBLE PRECISION argument.

An intrinsic function is invoked like any other function, using either the intrinsic or the generic name. If the generic name is used, such as CSIN for COMPLEX sine, FORTRAN determines which intrinsic function to use by matching it to the data type of the argument. F77L does not issue a warning when multiple arguments of different types are used with intrinsic functions.

If an intrinsic function name is used as an actual argument in a CALL statement or external function reference, the intrinsic function name must be listed in an INTRINSIC statement (see Section 6.8).

The intrinsic functions are listed in the following tables that group the functions by common tasks. The tables give both the generic and intrinsic names of each function, the number of arguments required, the argument types and the function types. Below the usage definition listing for certain functions are numbers referring to notes on intrinsic function semantics and ranges at the end of this chapter. The following program demonstrates the use of an intrinsic function:

```
PROGRAM main
PRINT *, sqrt(2.0)
PRINT *, sqrt(2.0D0)
END
```

This program prints:

```
bbb1.414214
bbb1.4142135623730951
```

11.1 Numeric Functions

Usage Definition	Generic Name	Intrinsic Name	Number of Arguments	Argument Type	Function Type
Type conversion					
conversion	INT	–	1	numeric**	INTEGER
to INTEGER		INT*	1	REAL	INTEGER
See Note 1		IFIX*	1	REAL	INTEGER
		IDINT	1	DOUBLE	INTEGER
		–	1	COMPLEX	INTEGER
	INT2	–	1	numeric	INTEGER*2
	INT4	–	1	numeric	INTEGER*4
conversion	REAL	FLOAT	1	INTEGER	REAL
to REAL		–	1	REAL	REAL
		SNGL	1	DOUBLE	REAL
		REAL	1	COMPLEX	REAL
conversion to DOUBLE PRECISION	DBLE	–	1	numeric	DOUBLE
conversion to COMPLEX See Note 2	CMPLX	–	1 or 2	numeric	COMPLEX
conversion to COMPLEX*16	DCMPLX	–	1 or 2	numeric	COMPLEX*16
Truncation					
INT(a)	AINT	AINT	1	REAL	REAL
See Note 1		DINT	1	DOUBLE	DOUBLE
Nearest whole number					
INT(a+.5) if a > 0	ANINT	ANINT	1	REAL	REAL
INT(a–.5) if a < 0		DNINT	1	DOUBLE	DOUBLE
Nearest integer					
INT(a+.5) if a > 0	NINT	NINT	1	REAL	INTEGER
INT(a–.5) if a < 0		IDNINT	1	DOUBLE	INTEGER
		I2NINT	1	REAL/DOUBLE	INTEGER*2

For arguments of type REAL, INT and IFIX have the same effect.
For conversion between integers and ASCII characters, see Section 11.5.

** Numeric refers to either a physical number constant or variable name.

The intrinsic function notes are on pages 11-12 to 11-15.

11.1 Numeric Functions (Continued)

Usage Definition	Generic Name	Intrinsic Name	Number of Arguments	Argument Type	Function Type
Absolute value					
|a|	ABS	IABS	1	INTEGER	INTEGER
		ABS	1	REAL	REAL
		DABS	1	DOUBLE	DOUBLE
See Note 3		CABS	1	COMPLEX	REAL
		I2ABS	1	INTEGER*2	INTEGER*2
See Note 3		CDABS	1	COMPLEX*16	DOUBLE
Remaindering					
a1 MODULO a2 or	MOD	MOD	2	INTEGER	INTEGER
a1−INT(a1/a2)*a2		AMOD	2	REAL	REAL
See Note 4		DMOD	2	DOUBLE	DOUBLE
		I2MOD	2	INTEGER*2	INTEGER*2
Transfer of sign					
|a1|if a2 ≥0	SIGN	ISIGN	2	INTEGER	INTEGER
−|a1| if a2 < 0		SIGN	2	REAL	REAL
See Note 5		DSIGN	2	DOUBLE	DOUBLE
		I2SIGN	2	INTEGER*2	INTEGER*2
Positive difference					
a1−MIN(a1,a2)	DIM	IDIM	2	INTEGER	INTEGER
		DIM	2	REAL	REAL
		DDIM	2	DOUBLE	DOUBLE
		I2DIM	2	INTEGER*2	INTEGER*2
Double precision product					
a1*a2		DPROD	2	REAL	DOUBLE
Select largest value					
MAX(a1,a2,...)	MAX	MAX0	2+	INTEGER	INTEGER
		AMAX1	2+	REAL	REAL
		DMAX1	2+	DOUBLE	DOUBLE
		AMAX0	2+	INTEGER	REAL
		MAX1	2+	REAL	INTEGER
		I2MAX0	2+	INTEGER*2	INTEGER*2
Select smallest value					
MIN(a1,a2,...)	MIN	MIN0	2+	INTEGER	INTEGER
		AMIN1	2+	REAL	REAL
		DMIN1	2+	DOUBLE	DOUBLE
		AMIN0	2+	INTEGER	REAL
		MIN1	2+	REAL	INTEGER
		I2MIN0	2+	INTEGER*2	INTEGER*2
Imaginary part of a COMPLEX argument					
y in (x,y) pair	AIMAG	AIMAG	1	COMPLEX	REAL
See Note 6		DIMAG	1	COMPLEX*16	DOUBLE
Conjugate of a COMPLEX argument					
(ar,−ai)	CONJG	CONJG	1	COMPLEX	COMPLEX
See Note 6		DCONJG	1	COMPLEX*16	COMPLEX*16

11.2 Transcendental Functions

Usage Definition	Generic Name	Intrinsic Name	Number of Arguments	Argument Type	Function Type
Square root					
(a)**(1/2)	SQRT	SQRT	1	REAL	REAL
See Note 7		DSQRT	1	DOUBLE	DOUBLE
		CSQRT	1	COMPLEX	COMPLEX
		CDSQRT	1	COMPLEX*16	COMPLEX*16
Exponential					
e**a	EXP	EXP	1	REAL	REAL
		DEXP	1	DOUBLE	DOUBLE
		CEXP	1	COMPLEX	COMPLEX
		CDEXP	1	COMPLEX*16	COMPLEX*16
Natural log					
LOG(a)	LOG	ALOG	1	REAL	REAL
See Note 8		DLOG	1	DOUBLE	DOUBLE
		CLOG	1	COMPLEX	COMPLEX
		CDLOG	1	COMPLEX*16	COMPLEX*16
Common log					
LOG10(a)	LOG10	ALOG10	1	REAL	REAL
See Note 8		DLOG10	1	DOUBLE	DOUBLE

The intrinsic function notes are on pages 11-12 to 11-15.

* The result of a COMPLEX function is the principal value.

11.3 Trigonometric Functions

Usage Definition	Generic Name	Intrinsic Name	Number of Arguments	Argument Type	Function Type
Sine					
sine(a)	SIN	SIN	1	REAL	REAL
See Note 9		DSIN	1	DOUBLE	DOUBLE
		CSIN	1	COMPLEX	COMPLEX
		CDSIN	1	COMPLEX*16	COMPLEX*16
Cosine					
cosine(a)	COS	COS	1	REAL	REAL
See Note 9		DCOS	1	DOUBLE	DOUBLE
		CCOS	1	COMPLEX	COMPLEX
		CDCOS	1	COMPLEX*16	COMPLEX*16
Tangent					
tangent(a)	TAN	TAN	1	REAL	REAL
See Note 9		DTAN	1	DOUBLE	DOUBLE
Arc cosine					
arccosine(a)	ACOS	ACOS	1	REAL	REAL
See Note 10		DACOS	1	DOUBLE	DOUBLE
Arc sine					
arcsine(a)	ASIN	ASIN	1	REAL	REAL
See Note 11		DASIN	1	DOUBLE	DOUBLE
Arc tangent					
arctan(a)	ATAN	ATAN	1	REAL	REAL
See Note 12		DATAN	1	DOUBLE	DOUBLE
arctan(a1/a2)	ATAN2	ATAN2	2	REAL	REAL
See Note 12		DATAN2	2	DOUBLE	DOUBLE

The intrinsic function notes are on pages 11-12 to 11-15.

* All angles are expressed in radians.
The result of a COMPLEX function is the principal value.

11.4 Hyperbolic Functions

Usage Definition	Generic Name	Intrinsic Name	Number of Arguments	Argument Type	Function Type
Hyperbolic sine					
SINH(a)	SINH	SINH	1	REAL	REAL
		DSINH	1	DOUBLE	DOUBLE
Hyperbolic cosine					
COSH(a)	COSH	COSH	1	REAL	REAL
		DCOSH	1	DOUBLE	DOUBLE
Hyperbolic tangent					
TANH(a)	TANH	TANH	1	REAL	REAL
		DTANH	1	DOUBLE	DOUBLE

11.5 CHARACTER Functions

Usage Definition	Generic Name	Intrinsic Name	Number of Arguments	Argument Type	Function Type
ASCII comparison					
lexically greater than or equal	LGE	LGE	2	CHARACTER	LOGICAL
lexically greater than	LGT	LGT	2	CHARACTER	LOGICAL
lexically less than or equal	LLE	LLE	2	CHARACTER	LOGICAL
lexically less than	LLT	LLT	2	CHARACTER	LOGICAL
See Note 13 for all these functions					

The intrinsic function notes are on pages 11-12 to 11-15.

The intrinsic functions LGE, LGT, LLE and LLT compare two CHARACTER entities in accordance with the ASCII collating sequence. In FORTRAN their logical results are the same as those of the corresponding CHARACTER relational expressions. These intrinsic functions are provided to allow for portability of code to processors where other collating sequences may be used.

Usage Definition	Generic Name	Intrinsic Name	Number of Arguments	Argument Type	Function Type
Length					
length of argument	LEN	LEN	1	CHARACTER	INTEGER
See Note 14					
Location of a substring in a character sequence					
INDEX(a1,a2) returns starting position of "a2" within "a1"; "0" if no occurrence. See Note 15	INDEX	INDEX	2	CHARACTER	INTEGER
ASCII conversion					
INTEGER to ASCII	CHAR	CHAR	1	INTEGER	CHARACTER
ASCII to INTEGER	ICHAR	ICHAR	1	CHARACTER	INTEGER
See Notes 16 and 17					

The intrinsic function notes are on pages 11-12 to 11-15.

The functions ICHAR and CHAR convert a CHARACTER value to and from the INTEGER equivalent representation.

Example:

```
CHARACTER numb*3
numb = '100'
int = ICHAR(numb(3:3))                [int=48]
int = int+5                           [int=53)
numb(3:3) = CHAR(int)                 [numb='105']
```

converts the CHARACTER number from "100" to "105".

If the CHARACTER argument to ICHAR has a "len" greater than 1, only the leftmost character is used.

11.6 Nonstandard Functions

The following F77L functions are extensions to the standard. The generic name is the same as the intrinsic name for these functions. These function names cannot be passed as arguments.

11.6.1 Trailing Blanks Functions

NBLANK and CHARNB are functions that deal with the trailing blanks of CHARACTER arguments.

NBLANK returns a result of type INTEGER*2. The returned value indicates the position of the last nonblank character. The syntax is:

```
NBLANK(e)
```

Where:

> "e", a CHARACTER expression, is searched for the last nonblank character.

Example:

```
c='aƀbƀcƀdƀƀ'
PRINT *, c(1:NBLANK(c))
```

prints

```
aƀbƀcƀd
```

CHARNB returns a CHARACTER value from which trailing blanks have been removed. If the argument has only blank characters, a single blank character is returned. The syntax is:

```
CHARNB(e)
```

Where:

> "e" is a CHARACTER expression.

Example:

```
c='aƀbƀcƀdƀƀ'
PRINT *, CHARNB(c)
```

prints

```
aƀbƀcƀd
```

11.6.2 Random Number Generators

RND, RRAND and RANDS generate uniformly distributed pseudorandom numbers between 0.0 and 1.0. Their syntax is as follows:

RND() **RRAND()** **RANDS(x)**

RND and RRAND require no arguments. RANDS requires a REAL argument "x". One REAL value is returned each time any of these functions is invoked.

Normally, RRAND is used the first time a random number is needed in a given program. The function references the system clock to generate a pseudorandom value that is returned and a seed value that is saved for possible future reference by the RND function. If more random numbers are required during the course of the program, the RND function may be used to generate these without the necessity of a time-consuming reference to the system clock.

If you want a program to produce the same series of pseudorandom numbers from one execution to the next, the RANDS(x) function may be used. In this case you are supplying the seed for the first number produced so that the sequence of random numbers produced by any subsequent use of the RND function is the same as long as "x" is not changed. Note that the "/7" Compiler Option must be specified in either the F77L.FIG File or on the command line when testing for equality.

11.6.3 Boolean Functions

The following Boolean functions perform logical operations on INTEGER arguments:

IAND(i1,i2) IEOR(i1,i2) IOR(i1,i2) NOT(i)

These functions return either INTEGER or INTEGER*2 results, depending upon the type of the argument(s). The types of the two arguments must be the same. An INTEGER constant argument less than 2**64 is considered to be the same type as the other argument.

On a bit-by-bit basis, for the following values of the arguments ("i1" and "i2"), the results of the functions IAND, IEOR and IOR are:

i1	**0**	**0**	**1**	**1**
i2	**0**	**1**	**0**	**1**
IAND(i1,i2)	**0**	**0**	**0**	**1**
IEOR(i1,i2)	**0**	**1**	**1**	**0**
IOR(i1,i2)	**0**	**1**	**1**	**1**

The following Boolean function performs logical negation on an INTEGER argument:

NOT(i)

This function returns either an INTEGER or INTEGER*2 result in agreement with the type of the argument, "i".

On a bit-by-bit basis, the value of this function is:

I	**0**	**1**
NOT(i)	**1**	**0**

11.6.4 Shift Function

This function provides a logical shift:

ISHFT(i1,i2)

Where:

"i1" specifies the INTEGER or INTEGER*2 unsigned value to be shifted; and

"i2" specifies the shift count, i.e., the power of 2 by which "i1" is to be adjusted; if "i2" is greater than zero, then "i1" is shifted left "i2" places; if i2" is zero, then no shift occurs; if "i2" is less than zero, then "i1" is shifted right.

11.6.5 Address Functions

Three functions are provided in F77L to pass addresses as arguments to functions and subroutines:

SEGMENT(item) OFFSET(item) POINTER(item)

OFFSET and SEGMENT return INTEGER*2 results, and POINTER returns an INTEGER result, with the SEGMENT in the high-order 16 bits. The segment is adjusted to minimize the offset.

These functions are provided in F77L-EM/32 to pass addresses as arguments to functions and subroutines:

SEGMENT(item) OFFSET(item)

OFFSET returns an INTEGER*4 result and SEGMENT returns an INTEGER*2 result.

11.6.6 NARGS Function

NARGS is an INTRINSIC INTEGER function that returns the number of arguments passed to the subprogram in which it appears. The syntax of the function is:

NARGS()

11.7 Notes on Intrinsic Functions

The following notes list the syntax and ranges for intrinsic functions as defined in the ANSI X3.9 – 1978 FORTRAN 77 Standard. The intrinsic function tables in this chapter refer to these notes by number.

1. For $\underline{a}$ type of integer, int ($\underline{a}$) = $\underline{a}$. For $\underline{a}$ of type REAL or DOUBLE PRECISION, there are two cases:

   if $|\underline{a}| < 1$, int ($\underline{a}$) = 0

   if $|\underline{a}| \geq 1$, int ($\underline{a}$) is the integer whose magnitude is the largest integer that does not exceed the magnitude of $\underline{a}$ and whose sign is the same as the sign of $\underline{a}$. For example, int (–3.7) = –3

2. CMPLX may have one or two arguments. If there is one argument, it may be of type INTEGER, REAL, DOUBLE PRECISION or COMPLEX. If there are two arguments, they must both be of the same type and may be of type INTEGER, REAL or DOUBLE PRECISION.

   For $\underline{a}$ of type COMPLEX, CMPLX ($\underline{a}$) is $\underline{a}$. For $\underline{a}$ of type INTEGER, REAL or DOUBLE PRECISION, CMPLX ($\underline{a}$) is the COMPLEX value whose real part is REAL ($\underline{a}$) and whose imaginary part is zero.

   CMPLX ($\underline{a_1}$, $\underline{a_2}$) is the COMPEX value whose real part is REAL ($\underline{a_1}$) and whose imaginary part is REAL ($\underline{a_2}$)

3. A complex value, $\underline{z}$, is expressed as an ordered pair of REALs, ($\underline{ar}$, $\underline{ai}$), where $\underline{ar}$ is the real part and $\underline{ai}$ is the imaginary part; then

   $$ABS\ (\underline{z}) = (\underline{ar}^2 + \underline{ai}^2)^{1/2}$$

4. Remaindering: The result for MOD, AMOD and DMOD is undefined when the value of the second argument is zero.

5. Transfer of Sign: If the value of the first argument of ISIGN, SIGN or DSIGN is zero, the result is zero, which is neither positive or negative.

6. A complex value, z, is expressed as an ordered pair of REALs, (ar, ai), where ar is the real part and ai is the imaginary part.

7. Square Root: The value of the argument of SQRT and DSQRT must be greater than or equal to zero. The result of CSQRT is the principal value with the real part greater than or equal to zero. When the real part of the result is zero, the imaginary part is greater than or equal to zero.

8. Logarithms: The value of the argument of ALOG, DLOG, ALOG10 and DLOG10 must be greater than zero. The value of the argument of CLOG must not be (0.,0.). The range of the imaginary part of the result of CLOG is: $-\pi <$ imaginary part $\leq \pi$. The imaginary part of the result is π only when the real part of the argument is less than zero and the imaginary part of the argument is zero.

9. Sine, Cosine and Tangent: The absolute value of the argument of SIN, DSIN, COS, DCOS, TAN and DTAN is not restricted to be less than 2π.

10. Arccosine: The absolute value of the argument of ACOS and DACOS must be less than or equal to one. The range of the result is: $0 \leq$ result $\leq \pi$.

11. Arcsine: The absolute value of the argument of ASIN and DASIN must be less than or equal to one. The range of the result is: $-\pi/2 \leq$ result $\leq \pi/2$.

12. Arctangent: The range of the result for ATAN and DATAN is: $-\pi/2 \leq$ result $\leq \pi/2$. If the value of the first argument of ATAN2 or DATAN2 is positive, the result is positive. If the value of the first argument is zero, the result is zero if the second argument is positive and π if the second argument is negative. If the value

of the first argument is negative, the result is negative. If the value of the second argument is zero, the absolute value of the result is $\pi/2$. The arguments must not both have the value zero. The range of the result for ATAN2 and DATAN2 is: $-\pi <$ result $\leq \pi$.

13. If the operands for LGE, LGT, LLE and LLT are of unequal length, the shorter operand is processed as if it were extended on the right with blanks to the length of the longer operand.

    The F77L CHARACTER type permits values in the range 0 to 255, inclusive, so that the collating sequence referred to below is this range which extends the ASCII character set defined in **Appendix M**.

    LGE ($\underline{a}_1$, $\underline{a}_2$) returns the value TRUE if $\underline{a}_1 = \underline{a}_2$ or if $\underline{a}_1$ follows $\underline{a}_2$ in the ASCII character set collating sequence, and otherwise returns the value FALSE.

    LGT ($\underline{a}_1$, $\underline{a}_2$) returns the value TRUE if $\underline{a}_1$ follows $\underline{a}_2$ in the collating sequence, and otherwise returns the value FALSE.

    LLE ($\underline{a}_1$, $\underline{a}_2$) returns the value TRUE if $\underline{a}_1 = \underline{a}_2$ or if $\underline{a}_1$ precedes $\underline{a}_2$ in the collating sequence, and otherwise returns the value FALSE.

    LLT ($\underline{a}_1$, $\underline{a}_2$) returns the value TRUE if $\underline{a}_1$ precedes $\underline{a}_2$ in the collating sequence, and otherwise returns the value FALSE.

14. The value of the argument of the LEN function need not be defined at the time the function reference is executed.

15. INDEX ($\underline{a}_1$, $\underline{a}_2$) returns an integer value indicating the starting position within the CHARACTER string $\underline{a}_1$ of a substring identical to string $\underline{a}_2$. If $\underline{a}_2$ occurs more than once in $\underline{a}_1$, the starting position of the first occurence is returned.

    If $\underline{a}_2$ does not occur in $\underline{a}_1$, the value zero is returned. Note that zero is returned if LEN ($\underline{a}_1$) < LEN ($\underline{a}_2$).

16. CHAR($\underline{i}$) returns the character in the $\underline{i}$th position of the processor collating sequence. The value is of type CHARACTER of length one. $\underline{i}$ must be an integer expression whose value must be in the range: $0 \le i \le 255$.

ICHAR(CHAR($\underline{i}$)) = $\underline{i}$	for $0 \le \underline{i} \le 255$.
CHAR(ICHAR($\underline{c}$)) = $\underline{c}$	for any character $\underline{c}$ capable of representation in the processor.

17. ICHAR provides a means of converting from an 8-bit CHARACTER to an integer.

    The value of ICHAR ($\underline{a}$) is an integer in the range: $0 \le$ ICHAR ($\underline{a}$) ≤ 255, where $\underline{a}$ is an argument of type CHARACTER of length one.

    For any characters $\underline{c_1}$ and $\underline{c_2}$ capable of representation in the processor, ($\underline{c_1}$ .LE. $\underline{c_2}$) is TRUE if and only if (ICHAR($\underline{c_1}$) .LE. ICHAR($\underline{c_2}$)) is TRUE, and ($\underline{c_1}$ .EQ. $\underline{c_2}$) is TRUE if and only if (ICHAR ($\underline{c_1}$) .EQ. ICHAR ($\underline{c_2}$)) is TRUE.

RESERVED FOR YOUR NOTES

12 SYSTEM SUBROUTINES

This chapter describes user-callable system subroutines provided as extensions to the FORTRAN Language. They are used like user-defined subroutines (see Section 10.3).

12.1 DOS Interface Subroutines

CALL TIME(result)

Returns the current system time-of-day in the form HH:MM:SS:hh in the CHARACTER datum "result", where HH (hour), MM (minute), SS (second), and hh (hundreths of a second) are two digits each. If "result" is not exactly 11 characters in length, the time is left-justified and truncated or space-filled as necessary.

CALL DATE(result)

Returns the current date in the form MM/DD/YY in the CHARACTER datum "result", where MM (month), DD (day) and YY (year) are two digits each. If "result" is not exactly eight characters in length, the date is left-justified and truncated or space-filled as necessary.

CALL GETCL(result)

Returns the command line the user entered following the command to execute a program in the CHARACTER datum "result", left-justified and truncated or space-filled as necessary. The text following the command may be ug to 127 characters long. It begins with the first delimiter following the command that names the file that is to be executed.

CALL INTRUP(intary, ntrup)

This subroutine accesses internal interrupts from FORTRAN code. Note the differences between the F77L and F77L-EM/16 and the F77L-EM/32 implementations.

F77L and F77L-EM/16

The array "intary" is an INTEGER*2 array of nine elements which corresponds to the registers AX, BX, CX, DX, DS, ES, DI, SI and flags, in that order. The registers are loaded (except flags) from the array before the interrupt is executed. The registers are stored back into the array after the interrupt is finished. "ntrup" is the INTEGER*2 interrupt number.

F77L-EM/32

The array "intary" is an INTEGER*4 array of nine elements which corresponds to the registers EAX, EBX, ECX, EDX, DS, ES, EDI, ESI and flags, in that order. The registers are loaded (except flags) from the array before the interrupt is executed. The registers are stored back into the array after the interrupt is finished. "ntrup" is the INTEGER*2 interrupt number.

The remaining intrup instructions are the same for all products.

Internal DOS and BIOS functions can be executed using this subroutine. For more information see the technical reference manuals for DOS and your computer.

To check the flags after returning from "intrup", use the following code:

```
IF(iand(intary(9), flag) .NE.0) THEN...
```

where flag specifies which processor flag to test

Carry	**- flag = 1**
Parity	**- flag = 4**
Auxiliary Carry	**- flag = 16**
Zero	**- flag = 64**
Sign	**- flag = 128**
Trap	**- flag = 256**
Interrupt Enable	**- flag = 512**
Direction	**- flag = 1024**
Overflow	**- flag = 2048**

For further information on flags, see technical documents for the processor you are using.

CALL EXIT(ilevel)

This subroutine allows the FORTRAN Programmer to terminate a program with ERRORLEVEL of DOS set to "ilevel" which is of type INTEGER.

Example:

```
CALL EXIT(5)
```

CALL SYSTEM(comnd)

This subroutine passes a CHARACTER expression "comnd" to DOS to be executed as if it had been typed at the console. The maximum length of "comnd" is 77 characters.

Example:

```
CALL SYSTEM('>current.dir dir')
```

puts a listing of the current directory into the file "current.dir", which could then be OPENed and manipulated by the program.

NOTES: The SYSTEM subroutine needs to load a new copy of the command processor COMMAND.COM in order to operate. DOS expects COMMAND.COM to be on the disk drive from which the computer was booted. If the computer was booted from a floppy disk, then a copy of COMMAND.COM must be available on that drive.

CALL TIMER(iticks)

The count of hundredths of seconds elapsed since midnight is returned in "iticks", an INTEGER*4 variable or array element. This routine is useful for optimizing algorithms by checking the "iticks" before and after executing a section of code, then finding the difference.

12.2 F77L Input/Output Subroutines

CALL IOSTAT_MSG(iostat,message)

This subroutine is passed a value, "iostat", that was created by the IOSTAT= specifier in an input/output statement. The CHARACTER variable "message" will be assigned the message text for the error detected by the input/output statement. If the runtime error file "F77L.EER" is not found on the path or if "iostat" does not contain a valid number, "message" will be assigned blanks. The variable "message" should be at least 64 characters long; "iostat may be either INTEGER or INTEGER*2. If the message cannot be read because: no file handles are available, the error file is not on the path or the message number is invalid, then the message returned will be blank.

Example:

```
      CHARACTER *64 message
      INTEGER*2 iostat
      OPEN (UNIT=1, FILE='NOFILE',STATUS='OLD',
     &IOSTAT=iostat)
      IF (iostat.NE.0) THEN
          CALL IOSTAT_MSG (iostat, message)
          PRINT *, message
      ENDIF
```

CALL FLUSH(iunit)

Empties the buffer associated with the output file designated by INTEGER datum "iunit" by writing the buffer to its file. This does not flush the DOS file buffers.

CALL PROMPT(message)

The CHARACTER datum "message" becomes the sequence which appears on the console each time the program requests keyboard input. If a prompt message is not defined with this subroutine, the F77L runtime package will not prompt the user when input is requested. The maximum length of a prompt string is 80 characters.

CALL PRECFILL (filchr)

If the format specification for a floating point number (REAL, DOUBLE PRECISION or COMPLEX) requires printing more significant digits than the source value is capable of representing, those positions for which there are no significant digits available are filled with a fill character (a zero at program startup). PRECFILL changes the fill character from zero to any character the user desires (e.g., "*" or " "). PRECFILL must be called with a CHARACTER argument ("filchr") whose first character becomes the new precision overflow fill character until changed by another call to PRECFILL.

Example:

```
        REAL X
        DATA X/123456.7/
10      FORMAT (X,F20.10)
        WRITE (*,10) X
        CALL PRECFILL ('*')
        WRITE (*,10) X
        END
```

produces:

```
123456.7000000000
123456.7*********
```

12.2.1 Arithmetic Exception Subroutines

CALL INVALOP(lflag)
CALL OVEFL(lflag)
CALL UNDFL(lflag)
CALL DVCHK(lflag)

These subroutines allow users to continue processing after an NDP (8087, 80287 or 80387) invalid operation, arithmetic overflow, underflow or divide-by-zero, respectively. The routine must be called once for setup, then each subsequent call sets the LOGICAL (*4 or *1) argument "flag" "TRUE" if the condition has occurred and "FALSE" otherwise. Use of the results of calculations which cause these conditions may cause the invalid operation error message from the NDP. Refer to an NDP technical manual for further information.

CALL UNDER0(lflag)

In many programming applications, if the result of an arithmetic calculation produces a number whose magnitude is too small to represent in the data storage cell (underflow), it is best to consider the result to be zero. This subroutine controls a feature which causes an underflow on a store into memory operation to store a zero value instead of reporting the underflow. If the LOGICAL (*4 or *1) argument, "lflag", is "TRUE", the feature is turned ON; if "FALSE", it is turned OFF.

It is on the store into memory operation that probably 99.99% of the possible underflows in an F77L program would occur. While it is theoretically possible to have an underflow on an intermediate calculation, it is extremely unlikely since the magnitude of the number would have to be less than 3.4 E – 4932.

"UNDER0" cannot be used in a program that also uses the system subroutine "UNDFL". When compiling with "/E" for emulation, any underflow automatically results in a 0 being stored and continuation of execution.

12.2.2 Miscellaneous Subroutines

CALL ENSSTK (Istack)

Programs which call subroutines written in languages other than F77L and linked with an F77L Main Program are allocated at least 240 bytes of stack storage. If more stack than that is required, the subroutine ENSSTK can be called to ensure that at least the specified number of bytes of stack is available to all non-F77L routines called by F77L. The change in amount of overhead stack storage remains in effect until program termination.

The argument "istack" is INTEGER*2.

CALL ENSSTK(istack)

Example:

```
CALL ENSSTK(1000)
```

causes 1000 bytes of stack to be reserved for all subsequent non-F77L routines invoked by the program.

CALL ERROR(message)

Prints the contents of the CHARACTER datum "message" at the console, with a subprogram traceback. This subroutine does not terminate the program.

_USERNIT and _USERTRM

You may use these two subroutines for doing some set-up before or clean-up after executing an F77L program. F77L invokes the assembly language routine _USERNIT before starting the user code, and _USERTRM just before exiting to DOS. Register AL is set with the intended error code to return to DOS on entry to _USERTRM; the value in AL at the end of _USERTRM will be passed to DOS as an error code.

If routines with these names are not linked with an F77L program, dummy routines from the F77L Library will be executed instead.

RESERVED FOR YOUR NOTES

INSTALLATION and OPERATION A

PROGRAMMING HINTS B

INSTALLATION & OPERATIONS

INSTALLATION &
OPERATIONS

A INSTALLATION and OPERATION

This appendix describes how to install and configure the F77L Language System and how to compile link and execute FORTRAN programs. F77L, as delivered, is configured to compile FORTRAN 77 Standard-Format Source Files and assumes the source filename extension, ".FOR".

A.0.1 System Requirements

F77L requires an 8086-, 8088-, 80286- or 80386-based IBM, Compaq or fully compatible computer with at least 256KB of RAM and DOS Version 2.0 or higher. Use of the /K Option requires an 80386-based computer with a Weitek 1167 or 3167 (Abacus) Math Coprocessor in addition to a virtual-8086 driver (e.g. "CEMM.SYS" for Compaqs). Refer to the "Weitek Considerations" section of the "READ.ME" File for up-to-date information on specific drivers. Use of the Lahey Blackbeard Editor requires 384KB of RAM.

A.0.2 Typographic Conventions

When the instructions in these appendices ask you to enter text, the required input is highlighted in **Boldface** type. You are required to press either the **RETURN** or **ENTER** key after the input to perform the desired operation. Any screen messages or prompts are shown in ***Bold Italic*** type.

- **< >** An item within angle brackets is to be replaced by something that will be described immediately following.
- { } Braces are used to group items together.
- [ ] Anything within square brackets is optional.
- | The vertical line is a logical "or". The construct {B|T} means that either B or T must be specified, but not both.

... The ellipsis means that the previously defined pattern can be repeated several times.

A.1 The F77L Install Procedure

The install procedure loads the F77L FORTRAN Language System files into a subdirectory from the distribution diskettes for a configuration that you select by responding to the prompts. A corresponding procedure that follows allows you to run a simple FORTRAN program. Subdirectories may be utilized to better manage the file system. We recommend using this install procedure to select the language system configuration you prefer.

By default the install procedure creates the subdirectory "\F77L" in which the F77L Language System Files are located and the "\F77L\SUPL", and "\F77L\DEMO" subdirectories where supplemental and demo programs are stored.

The install procedure also supports installation for systems with two floppy drives and no hard disk although we recommend having a hard disk in your system.

The install procedure does not automatically modify your "AUTOEXEC.BAT" File's PATH Command or any "CONFIG.SYS" File settings. Insert the following command into the "AUTOEXEC.BAT" File on the boot disk (or modify the PATH Command already there to include the text following the "="):

PATH= C:\F77L

After the computer is rebooted, the PATH Command in the "AUTOEXEC.BAT" File is in effect, causing DOS to find the files ending in ".EXE", ".COM", or ".BAT" in the Subdirectory "\F77L". The F77L Compiler and runtime system will also search the path for the Error (".CER" and ".EER"), Configuration (".FIG") and Patch (".FIX") Files. When using the PATH Command shown above, F77L searches two directories: first the current, then "\F77L".

To install the F77L Language System, insert the F77L Disk #1 into Drive A: and type:

A:INSTALL

Then answer the prompts to choose the language system configuration desired.

The install procedure runs from any floppy drive and also allows you to specify alternate destination drives and subdirectories. If you choose a subdirectory other than "\F77L", change the PATH Command accordingly.

A.1.1 Testing the Installation

To test the installation, change to the subdirectory containing "DEMO.FOR" and compile, link, and run it by typing:

RUN DEMO

This invokes the batch file "RUN.BAT" in the subdirectory "F77L". This will not work if there is another file with the prefix "RUN" and an extension of ".BAT", ".COM" or ".EXE" in the current directory or anywhere else on the path. Be sure to specify a path to the Lahey OPTLINK Linker in your PATH Command.

NOTE: If you do not want to use the install procedure you may use the DOS Copy Command to install the F77L FORTRAN Language System. See the "READ.ME" File for a complete list of files on the distribution diskettes. The "READ.ME" File list describes the function of each file and has asterisks next to the required files for each element of the language system to help you determine the minimum installation configuration.

A.1.2 Setting Environment Variables

Some Lahey Language System development tools can use DOS "SET" Environment Variables to enhance their operations. See your DOS Manual for information on the SET Command. Follow these instructions to set variables in your "AUTOEXEC.BAT" File for each tool listed below.

Lahey Blackbeard Variables

Add the following lines to your "AUTOEXEC.BAT" File:

SET LBDIR=d:\F77L — specifies the directory containing the Editor Files
SET LBCFG=d:\F77L\LB.CFG — specifies the path of the Configuration File
SET LBKEY=d:\F77L\LB.KEY — specifies the path of the Key Binding File
SET LBPST=d:\F77L\LB.PST — specifies the path of the Paste File
SET LBLNG=d:\F77L\LB.LNG — specifies the path of the Language Expansion File

Where:

"d:" is your hard disk; and

"\F77L" is the subdirectory that contains Lahey Blackbeard after installation.

NOTE: See the **System Configuration** section in the Lahey Blackbeard on-line help system for more information.

Lahey MAKE Variables

Add the following line to your "AUTOEXEC.BAT" File:

SET INIT=d:\F77L — specifies the directory containing "MAKE.INI", "MAKEFILE.PRG", and "MAKEFILE.LIB".

Where:

"d:" is your hard disk; and

"\F77L" is the subdirectory that contains Lahey MAKE after installation.

NOTE: See sections D.4 — D.5, D.17, and D.26 for more information on environment variables and Lahey MAKE.

Lahey OPTLINK Variables

Add the following lines to your "AUTOEXEC.BAT" File:

```
SET  LIB=[variables]
SET  TMP=[variables]
SET  OPTLINK=[variables]
```

Where the [variables] are described below.

Both the "LIB" and "TMP" variables may be configured for either:

- strict Microsoft Link compatibility, in which they specify only paths, or

- extended capability, in which case they specify paths if a trailing backslash "\" is present or files if it is absent.

The following description applies to the extended capability configuration.

NOTE: If configured for strict compatibility, presence or absence of the trailing backslash makes no difference; in either case, everything is taken to be a path.

LIB= This environment variable (if present) supplies paths to search for libraries (must have trailing backslash), or library files themselves (no trailing backslash). Note that this differs from MS-LINK usage.

TMP= This environment variable (if present) supplies a path (with trailing "\") used by Lahey OPTLINK for its virtual memory file. For good performance, this should be a RAM disk. For best performance, install an EMS driver instead.

OPTLINK= This environment variable (if present) supplies any default switch settings you would like. See section F.2.2 for information on Lahey OPTLINK switches.

For example, to always select option /IGNORECASE as the default, the following command could be used:

SET OPTLINK=/IG

The Lahey OPTLINK environment variables can be overridden by command-line input and every switch has an opposite, so all can be changed.

A.2 Developing in FORTRAN

The Lahey F77L FORTRAN Language System contains a complete set of powerful tools to edit, compile, link, debug, and optimize FORTRAN programs. The general steps involved in using F77L to develop a program are:

1. create a source file using the Lahey Blackbeard editor;

2. create a Make File using Lahey MAKE for multi-source file programs;

3. compile the source file to create an object file;

4. link the object file using Lahey OPTLINK to create an ".EXE" file;

5. execute the linked program; and

6. debug and optimize the program using SOLD and the Lahey Profiler.

A.3 Running a FORTRAN Program

The structure and syntax of a FORTRAN source file are given in **Chapter 1** and **Chapter 10**. In general, an executable program consists of a main program and zero or more subprograms. An example of a source file that contains one subprogram unit is as follows:

```
* Program, demonstrate MAIN + subprogram
  PROGRAM main
  CALL sub
  END
  SUBROUTINE sub
  PRINT *,"In sub, Hi there!"
  END
```

To execute a FORTRAN source program, the following steps must occur. First, the compiler translates a source file that contains one or more FORTRAN subprograms into an object file that must then be processed with the Lahey OPTLINK Linker. The Lahey OPTLINK Linker combines the object file(s) created by the compiler, and those runtime routines required to execute the compiled code (gathered from "F77L.LIB") into an ".EXE" file that may then be executed.

The compile, link and execute command file, "RUN.BAT", will run a FORTRAN program that is contained in a single source file. The source filename should be entered without an extension. The compiler will add one of the assumed extensions (either ".FOR" or ".F"). "RUN.BAT" can pass source filename extensions and command-line options to the compiler. To do this, type a space after the filename, and then type the extension and option(s).

Example:

RUN DEMO .FFF/R/NL

compiles the source file "DEMO.FFF" with the options "R" and "NL". To see an example of the steps required to compile, link and run a program, use the DOS "TYPE" Command on the file "RUN.BAT".

NOTE: DOS always reserves three files: STDIN, STDOUT, and STDERR, so the number of available file handles will always be three less than the "FILES=" setting in your "CONFIG.SYS" File. DOS allows, as a default, at most eight files open at one time, including these DOS reserved files. This limits the number of nested INCLUDE files in a FORTRAN program. To increase the number of files that can be opened, see "Configuring Your System" in the DOS Manual.

A.4 Compiling a FORTRAN Program

The F77L Compiler command-line syntax is the following:

```
F77L source[.ext] [,][object] [,][list] [,][sold] [/[N]switch]...
```

Where:

"source" is the name of the source file to be compiled.

".ext" is the source filename extension. If not specified, the compiler assumes an extension determined by the /F Option in the compiler configuration file, "F77L.FIG", described below. The extension ".FOR" is assumed for standard format source files (/NF). The extension ".F" is assumed for free format source files (/F). To specify a source file without an extension, terminate the source filename with a period.

"object" is the file to which compiler results are written. If "object" is not specified, the compiled source is written to "SOURCE.OBJ".

"list" is the optional file to which hardcopy and cross reference output is written. If "list" is not specified, the hardcopy and cross reference output is written to "SOURCE.LST". Specifying a listing filename will not create a listing file unless either the /X or /H Compiler Option is specified.

"sold" is the optional file to which the Source On-Line Debugger (SOLD) output is written. If "sold" is not specified, SOLD output is written to "SOURCE.SLD". The extension to the "sold" filename must be ".SLD".

"Source", "object", "list" and "sold" may include a drive and/or subdirectory pathname in DOS Format.

"/[N]switch..." is one or more of the following compiler options (in any order):

/0	- IMPLICIT NONE in effect
/N0	- Standard implicit typing in effect
/2	- Generate 80286 code
/N2	- No 80286 code generated
/3	- Generate 80386 Code
/N3	- No 80386 code generated
/7	- Load NDP (8087, 80287 or 80387) nonINTEGER operands from memory
/N7	- NDP nonINTEGER operands may be kept in the NDP between statements
/A	- Adjustable arrays are <65280 bytes
/NA	- Adjustable arrays can be >65280 bytes
/A1	- Arrays dimensioned (1) are assumed to be less than 65280 bytes
/NA1	- Arrays dimensioned (1) may be any size up to, but not including, 2**20 bytes.
/B	- Bounds check array subscripts
/NB	- No bounds checking
/C	- Output warning messages indicating statements that do not conform to the FORTRAN 77 Standard
/NC	- Suppress the output of warning messages indicating nonconformances to the FORTRAN 77 Standard
/D	- Direct files are processed without headers
/ND	- Direct files require F77L-generated headers
/E	- Emulate 8087, 80287 or 80387 instructions
/NE	- Require an 8087, 80287 or 80387 coprocessor
/F	- Free-format source file
/NF	- Standard-format source file
/H	- Hardcopy (source listing) output
/NH	- No hardcopy (listing) output
/I	- Subprogram interface check
/NI	- No subprogram interface check

/K	- Generate Weitek 1167 and 3167 Code
/NK	- No 1167 and 3167 code generated
/L	- Line-number table
/NL	- No line-number table
/O	- Output compiler options
/NO	- No options output
/P	- Protect constant arguments
/NP	- No protection
/R	- Remember (SAVE) subprogram local variables and arrays
/NR	- No remembering local items
/S	- Generate SOLD information file
/NS	- No SOLD information file generated
/T	- Type; default length of INTEGER type is 2, LOGICAL type is 1
/NT	- Default length of INTEGER is 4, LOGICAL is 4
/U	- Create unformatted sequential files compatible with F77L Version 2.22 and earlier
/NU	- Faster processing of unformatted sequential I/O
/W	- All warning messages output
/NW	- Not all warning messages output
/X	- Cross-reference listing
/NX	- No cross-reference listing
/Z1	- Perform production code optimizations (limits SOLD and P77L Profiler interfaces)
/NZ1	- No optimizations that would limit SOLD and P77L Profiler interfaces

Typing "**F77L**" followed by a carriage return (**ENTER**) on the command line causes the compiler to display a synopsis of the compiling options.

Examples:

F77L TEST

causes the compiler ("F77L.EXE") to translate "TEST.FOR" into a relocatable file "TEST.OBJ" (assuming no FATAL or ABORT errors). This works when the files "F77L.EXE", "F77L.CER", "F77L.FIX", "F77L.FIG" and "TEST.FOR" are all in the current directory on the default drive. Also, all required F77L files must be in a directory that is in the path.

F77L B:DEMO.FOR

directs the compiler to read the source file, "DEMO.FOR", from the B: Drive and to create "DEMO.OBJ" on the B: Drive (assuming no FATAL or ABORT errors).

The command:

F77L \PROJCTX\SRC\SUB1, \PROJCTX\OBJ\SUB1

causes F77L to compile "SUB1.FOR" in subdirectory "SRC" of subdirectory "PROJCTX" into "SUB1.OBJ" in subdirectory "OBJ" of "PROJCTX".

A.4.1 Errors In Compilation

As the source file is compiled, F77L prints the line number and first source statement of each program unit to the console. If any errors are detected, error messages are printed following the first statement. These messages occur by program unit and include the source filename, a line number (which agrees with the line number representation of most ASCII editors), error message text, and when possible, the source text.

Three types of error messages are output: **ABORT**, **FATAL** and **WARNING**.

ABORT	messages indicate that it is not practical to continue compilation.
FATAL	messages indicate that the compilation will continue, but no object file will be generated.
WARNING	messages indicate probable programming errors that are not serious enough to prevent execution.

The message syntax depends upon where in the compiler the error is detected. Some errors are collected over a program unit and output after all usages are known (e.g., used but not assigned). Other error messages are output as they are detected (e.g., syntax errors).

If no FATAL or ABORT errors are detected by the compiler, ERRORLEVEL is set to 0 (see DOS "IF" Subcommand of BATCH Files). FATAL errors detected by the compiler (improper syntax, for example) cause F77L to terminate the compiler process with ERRORLEVEL set to 4, and the object file IS NOT created.

The compiler uses one or two temporary files on the default drive: "F77LBNRY.TMP" and for some large subprograms "F77LSPL1.TMP". If the computer is rebooted while compiling a source file, these temporary files may be left in the directory.

A.5 Linking a FORTRAN Program

As mentioned above, a FORTRAN program, although translated into machine language, is still not directly executable. Before it is executed, the subprograms and the library routines that it references must be located, and an ".EXE" image of the program must be created. To perform these tasks, the system module "Lahey OPTLINK" is invoked in the following manner:

OPTLINK FP1[+FP2]...;

Where:

"OPTLINK" is the linker;

"FP[n]" is a compiled source file; and

"FP1.EXE" is the resulting executable file.

F77L directs the linker to look for "F77L.LIB" in the directory where "F77L.FIX" is found. If "F77L.LIB" is in a different directory, the full pathname of "F77L.LIB" must be specified to the linker. If the "F77L.LIB" Library is specified when linking programs with FORTRAN mains, the "F77L.LIB" must be the last library specified in the linker command line.

Examples:

OPTLINK DEMO;

links the F77L-compiled object file "DEMO.OBJ" with "F77L.LIB" to create "DEMO.EXE", where all files are in the current directory of the default drive.

OPTLINK B:DEMO,

links the F77L-compiled object file "DEMO.OBJ" on the B: Drive with "F77L.LIB", to create "DEMO.EXE" on the B: Drive.

CD\PROJCTX\OBJ
OPTLINK SUB1;

links the F77L-compiled object file "SUB1.OBJ" with "F77L.LIB" to create "SUB1.EXE". Files "SUB1.*" are in subdirectory "OBJ" of "PROJCTX", and files "F77L.*" are in subdirectory "F77L".

The compiled and linked ".EXE" file is executed by entering the filename:

SUB1

Errors detected by Lahey OPTLINK (e.g., undefined symbol) and runtime detected errors (e.g., subscript out of range) prevent complete execution of the program.

NOTE: To change the program, the source must be modified, recompiled and relinked. When the program is debugged, object files and the ".SLD" (SOLD) files may be deleted.

Refer to **Appendix F – Lahey Lahey OPTLINK** for complete descriptions of Lahey OPTLINK's operations.

A.5.1 External References

External references in F77L are resolved when the program is processed by Lahey OPTLINK. Thus, all subprograms referenced by a program need not exist when the program is compiled, but they should all exist before the program is actually linked. The link rules for subprograms are:

1. all program units named in the first sequence of filenames (before the first comma) of the Lahey OPTLINK Command are loaded into the resulting ".EXE" file whether or not they are referenced.

2. if any subprogram reference has not been defined, the libraries are searched for the unresolved reference. This step is repeated until all references are resolved or Lahey OPTLINK determines that the subprogram referenced does not appear in any of the libraries.

A.5.2 Stack and Heap Memory Management

<u>Standard F77L Model</u>

If you link your program without any of the special F77L ".OBJ" files, the runtime initialization code will take all available memory to use for stack and heap. The stack starts at high memory and grows downward. The heap starts at the end of the static data and grows upward towards the stack. The heap is used to allocate memory for file control blocks and buffers.

User-written assembly language code that needs to allocate memory can also use the heap (see **Appendix J**). The stack is used for non-saved local storage, passing arguments, compiler temporary storage and DOS. If you compile your program with the /NR Option, then the stack segment will be adjusted upon program unit entry, so that each program unit can have almost 64K bytes of local data on the stack. This scheme allows you to make very efficient use of the available memory without performing any analysis of stack or heap requirements. The program will only get a "memory exhausted error" if it is totally out of memory.

Fixed-Stack Model

If you want to limit the amount of memory your program uses or, if you want to spawn another process from within your F77L-compiled program, (see **SYSTEM** and **INTRUP** in section 12.1) then you should be using the fixed-stack model. To use the fixed-stack model, link the module "F77LFS.OBJ" with your other ".OBJ's" and specify /STACK:nnnn (see the **Lahey OPTLINK Appendix F**) if you need more than the default 2048 bytes of stack. Heap memory allocation is performed by the DOS memory allocator.

A.6 Compiler Options and Configuring

This section describes two features of the F77L Compiler:

1. how to specify compiler options on the command line; and

2. how to configure the compiler to minimize command line typing by modifying the configuration file, "F77L.FIG".

Compiler options are specified on the command line and in the file "F77L.FIG" using identical syntax (i.e., a slash, optionally followed by the letter "N" (to suppress an option), followed by alphanumeric characters designating the option).

When the compiler is invoked, the options in the "F77L.FIG" File are processed first, then those on the command line. Each specification of an option overrides its previous setting, so that an option specified on the command line overrides the setting in the configuration file.

A.6.1 The FIG Configuration-File Editor

Included with F77L is the "FIG.EXE" Configuration-File Editor and its corresponding help file, "FIG.HLP". To use the editor from any directory, type:

FIG

to create or modify an "F77L.FIG" Configuration File in the current directory. The editor displays a list of compiler options and descriptions similar to the "Output Compiler Options" screen.

Use the cursor-control keys to indicate the option you want to change, then press the **SPACEBAR** to toggle the option next to the indicator arrow ON or OFF.

You may also type:

H for help on the indicated option;

D to set the options as we distribute them;

R to restore the option settings that existed before you invoked "FIG.EXE";

L to modify the Listing Specifications for the "/H" and "/X" Options;

E to save the option settings and exit from "FIG.EXE"; and

Q to quit without saving the settings.

NOTE: "FIG.EXE" and "FIG.HLP" must be in the current directory or in a directory specified by the DOS PATH Command.

Options, described below in collating order, are explained with their meanings and consequences. At the beginning of each is the syntax for specifying the option on the command line or in the "F77L.FIG" File.

IMPLICIT NONE

Command line or F77L.FIG /[N]0

Specifying the /0 (zero) Option is equivalent to including an IMPLICIT NONE statement in each program unit of your source file. The /0 Option indicates that no implicit typing is in effect over the source file.

When /N0 is specified (the default condition), standard implicit typing rules as described in **Chapter 6** are in effect.

Generate 80286/80287 Code

Command line or F77L.FIG /[N]2

Specifying the /2 Option directs the compiler to generate instructions that are available only on 80286 processor (or the 80386 in 80286 mode). This will make programs a little smaller and faster, but they will not run on a "PC" or "PC/XT".

Specify /N2 to generate code for 8086 or 8088 PCs.

Generate 80386 Code

Command line or F77L.FIG /[N]3

Specifying the /3 Option directs the compiler to generate instructions which are available only on 80386 processors. This option will create programs with smaller and faster INTEGER*4 multiplies and divides, but they will not run on 80286-, 8088-, or 8086-based systems.

Specify /N3 to generate code for 8088- or 8086-based PCs.

NDP Memory Operands

Command line or F77L.FIG /[N]7

Specifying the /7 Option guarantees the consistency of REAL, DOUBLE PRECISION, COMPLEX and COMPLEX*16 calculations. Consider the following example:

```
1    X = S - T
2    Y = X - U
     .
     .
     .
3    Y = X - U
```

During compilation of statement 2, F77L recognizes that the value "X" is already in a register and does not cause the value to be reloaded from memory. At statement 3, the value "X" may or may not already be in a register, and so the value may or may not be reloaded accordingly.

Because the precision of the datum is greater in a register than in memory, a difference in precision at lines 2 and 3 may occur. Even though this technique produces efficient object code, some users prefer to maintain memory precision between statements.

Specify the /7 Option to choose the memory reference for non-INTEGER operands; that is, registers are reloaded. The default option, /N7, allows F77L to take advantage of the current values in registers, with possibly greater accuracy in low-order bits.

Adjustable-dimension Arrays Limited to 64KB

Command line or F77L.FIG /[N]A

When the /A Option is ON, all adjustable-dimension arrays are assumed to be less than 65280 bytes in length, so that the code generated to address the arrays will be smaller and more efficient.

Specifying /NA allows adjustable-dimension arrays to be any size up to, but not including, 2**20 bytes.

Arrays Dimensioned (1) Limited to 64KB

Command line or F77L.FIG /[N]A1

When the /A1 Option is ON, all arrays dimensioned (1) are assumed to be less than 65280 bytes in length, so that the code generated to address the arrays will be smaller and more efficient.

Specifying /NA1 allows arrays dimensioned (1) to be any size up to, but not including, 2**20 bytes.

BOUNDS Checking

Command line or F77L.FIG /[N]B

When the /B BOUNDS Option is ON, programs are protected from an array subscript being outside of its allocated area and destroying adjacent data.

Nonconforming Usage

Command line or F77L.FIG /[N]C

Specifying the /C Option causes the compiler to output warning messages when it encounters non-Standard FORTRAN 77 usage. These messages only indicate deviations from the FORTRAN Standard and not the exact nature of the deviation. This option will not detect runtime extensions; in particular, the operands of input/output specifiers, mixed-mode formatted input/output and calls of system subroutines. Non-standard type usage is only indicated in declarations. The /C Option will not issue warnings when it encounters the following:

blank lines with statement labels;

the "!" trailing comment delimiter; and

a tab in columns 1 through 6.

The /NC Option suppresses the output of warning messages when the compiler encounters non-Standard FORTRAN 77 usage.

Direct Files

Command line or F77L.FIG /[N]D

An F77L-generated direct file has a header to identify record length. This header is read by the OPEN and INQUIRE statements. If you do not want a header in a direct file so that the file will be the same as those generated by many other languages, /D eliminates F77L's header requirement.

Specify /ND to create direct files with headers.

NDP Emulation

Command line or F77L.FIG /[N]E

When the /E Option is ON, F77L-generated programs emulate the 8087, 80287, or 80387 NDP instruction set to eliminate the need for a math coprocessor in your system. The runtime system will still sense the presence of an NDP chip and use it if it is available.

When /NE is specified, F77L-generated programs require the presence of one of the coprocessors listed above in your computer. The programs will be smaller because the emulation code will not be included.

NOTES: The /E Option setting in the main program determines whether the emulator will be included in the executable file. It is therefore possible to compile subprograms with /E, then compile the main program with /NE to create a smaller executable file that requires the math coprocessor.

When you run on a machine without an NDP, the emulator sets underflows to 0 (zero). The UNDER0 subroutine is not necessary in this case, but can be included for compatibility if the program will also be run on machines with NDP's.

In order to use the Numeric Data Processor Emulation scheme that the Microsoft assemblers generate when they are using the /E Option, the following are to be considered reserved names, and are not valid for external functions, subroutines or entries:

FIARQQ, FICRQQ, FIDRQQ, FIERQQ, FISRQQ, FIWRQQ, FJARQQ, FJCRQQ, and FJSRQQ

FREE-FORMAT Source File

Command line or F77L.FIG /[N]F

If the /F FREE-FORMAT Option is ON, the compiler processes all source files as free-format (described in **Chapter 1**). The default filename extension for free-format source files is ".F".

If the FREE-FORMAT Option is OFF (/NF), the compiler expects the source to be in the form specified by the ANSI FORTRAN 77 Standard (described in **Chapter 1**). The default filename extension for standard-format source files is ".FOR".

A source filename extension, if specified, is capable of overriding the setting of the /F Option in the configuration file. If the source filename extension is ".FOR", the source file is processed as standard format. If the extension is ".F", the source file is processed as free format. If any other extension is present, the /F Option is not changed.

Specifying /NF on the command line overrides all previous settings of the /F Option.

The format of all INCLUDE files (see Section 5.2) is required to be the same format as the source file specified on the command line.

Examples:

If "F77L.FIG" consists of the following line:

```
/F/O/NB/I/NL/P/R/W
```

then free format is the default for the source file, and entering:

F77L END.F

causes the compiler to process the source file, "END.F", as a free-format source file. If no FATAL errors are detected, the compiler creates "END.OBJ" for the relocatable code.

NOTE: The ".F" extension is redundant since ".F" is the default extension when free-format source files are expected.

Entering:

F77L END

causes the compiler to process the source file, "END.F", as a free-format source file and, if no FATAL errors are detected, to create "END.OBJ" for the relocatable code. If /NF had been specified in "F77L.FIG", the compiler would process "END" as a standard-format source file.

Entering:

F77L END.FOR

causes the compiler to process the source file, "END.FOR", as a standard-format source file, and if no FATAL errors are detected, to create "END.OBJ" for the relocatable code.

NOTE: The ".FOR" extension is required since the compiler expects a free-format source file and ".F" is the default extension.

Hardcopy Listing Output

Command line or F77L.FIG /[N]H(spec=sval[,spec=sval]...)

Where:

"spec" is "W" for page width, "L" for page length or "I" for Listing INCLUDE files. The specifiers may be in any order.

For "W=sval", the page width specifier, "sval" is an unsigned decimal constant. The minimum page width is 60. If not specified, the page width is 128. The headings are designed to work on a 128 character line, so some of them may not be as easy to read on a shorter line.

For "L=sval", the page length specifier, "sval" is an unsigned decimal constant. The minimum page length is 20. If not specified, the page length is 57. The page length refers to the maximum number of lines F77L will print on a page. The printer must be adjusted to the paper page length.

For "I=sval", the INCLUDE listing specifier, "sval" is either "Y" to list INCLUDE files or "N" to not list INCLUDE files. If not specified, INCLUDE files will be listed.

When the Hardcopy Option is ON (/H), F77L generates a listing of the FORTRAN source program. The listing is generated in printable format.

In addition to a source file listing, the hardcopy shows the date and time of the compilation and the selected compiling options. See the **Output Compiler Options** Section.

To send the hardcopy output directly to the printer, use the DOS reserved device names "LPT1", "LPT2" or "PRN" to specify the printer on your F77L Command Line. Specifying "CON" is not recommended. For example, the command line:

F77L MYPROG.F,,PRN

causes F77L to send hardcopy output to the system printer.

On the other hand, the command line:

F77L MYPROG.F /H(W=100,L=50,I=N)

causes F77L to send hardcopy output to a file named "MYPROG.LST". The hardcopy output will have a width of 100, a length of 50 and INCLUDE files will not be listed. Specifying a listing file will not create a listing file unless either the /X or /H Compiler Option is specified.

INTERFACE Checking

Command line or F77L.FIG /[N]I

If the /I INTERFACE Option is ON, F77L generates code that checks subprogram interfaces (argument count, alternate return specifier count and subprogram class) at program unit entry time.

Specify /NI to suppress interface checking.

Generate Weitek 1167/3167 Code

Command line or F77L.FIG /[N]K

Specifying the /K Option directs the compiler to generate instructions that are available only on the Weitek 1167 and 3167 (Abacus) coprocessors. An 80386-based system with one of these coprocessors is required to execute programs compiled with this switch ON. An 80x87 coprocessor is not required and will not be used if present. This option also sets the /2 and /3 compiler options to ON and the /7 and /E options to OFF.

NOTES: The /K Option is incompatible with Microsoft Windows 286 or 386.

Due to design differences between the Weitek and INTEL 80x87 coprocessors, identical floating-point instructions executed on the two chips may produce results which differ in their low-order bits. For arithmetic operations involving repeated multiplications or divisions, results may differ to a greater degree. For this same reason, results from the intrinsic random number generation functions, RND(), RRAND(), and RANDS() will also differ.

Specify /NK to generate code for machines without a Weitek 1167 or 3167.

LINE-number Table

Command line or F77L.FIG /[N]L

When the /L LINE Option is ON, F77L generates 4 bytes for each executable statement of a subprogram: 2 for the line number and 2 for the OFFSET of the generated code. The line-number table is used to inform users of the statement that fails at execution time. The table is part of the 64K of code space available for each program unit. No execution time is required for this table.

OUTPUT Compiler Options

Command line or F77L.FIG /[N]O

When the /O Option is ON, the compiler options are displayed at the console immediately before the source file is compiled.

NOTE: The first line of "O" Option output (beginning with compiling) is controlled by the "O" Option in your F77L.FIG File. The details about option settings are output only output if the "O" Option is ON after processing the command line. If you reverse the "O" Option on the command line, you will get only one part of the output option text.

PROTECT Arithmetic Constant Arguments

Command line or F77L.FIG /[N]P

When the /P Option is ON, invoked subprograms are prevented from storing into arithmetic constants. Example:

```
CALL sub(5)
PRINT *, 5
END
SUBROUTINE sub(i)
i = i + 1
END
```

prints 5 using /P and 6 using /NP.

REMEMBER Local Variables and Arrays

Command line or F77L.FIG /[N]R

When the /R REMEMBER Option is ON, F77L allocates local variables in a compiler-generated SAVE area which is equivalent to having a SAVE statement in each subprogram. The values in arrays and variables which have been defined in one invocation of a subprogram are remembered in the next invocation.

NOTE: Although local items are SAVEd, they are not initialized unless they are explicitly initialized in the program unit.

If the REMEMBER Option is OFF (/NR), F77L allocates local variables on the stack each time the program unit is invoked. This requires less memory (since only those subprograms that are in the execution chain have local variables and arrays allocated) and less code (since stack storage is addressable at all times). Variables or arrays that are initialized (as in a DATA statement) are allocated in a compiler-generated common and are remembered.

If recursion is programmed, care is required to prevent the currently executing subprogram from destroying values required by an invoking version. SAVEd values could be destroyed since only one storage area is allocated for them.

NOTE: The RECURSIVE keyword deactivates the /R (REMEMBER) Option for the program unit in which it appears so that local variable space will be allocated for each invocation of the function or subprogram. Variables appearing in a SAVE statement or in a DATA initialization context will be statically allocated, so that values will be retained across function and subroutine invocations.

Generate SOLD Information File (.SLD)

Command line or F77L.FIG /[N]S

When the /S SOLD Option is ON, the compiler generates a file containing information for use by the Source On Line Debugger (SOLD). The name of the file will be the source filename with ".SLD" substituted for the filename extension, unless specified otherwise on the command line. (See **Compiling a FORTRAN Program**, Section A.4).

TYPE INTEGER*2 and LOGICAL*1

Command line or F77L.FIG /[N]T

When the /T TYPE Option is ON, the default length associated with INTEGER type is 2 bytes and with LOGICAL type is 1 byte. Conversely, the /NT Option sets default lengths for INTEGER type and LOGICAL type at 4 bytes. Lengths of 4 bytes conform to the ANSI Standard and are assumed if neither /T nor /NT is specified.

This option only affects:

> Constants whose absolute values are less than 32768, and
>
> INTEGER and LOGICAL items implicitly or explicitly typed without length specifications.

INTEGER and LOGICAL items that are implicitly or explicitly typed with length specifications are not affected.

NOTE: When using the /T switch, be sure to explicitly type all ASSIGN and Assigned GO TO statements as INTEGER*4.

An INTEGER constant used as an argument can be forced to length 2 or 4 with the INT2 and INT4 Intrinsic Functions.

Example:

```
CALL sub(INT4(5))
.
.
.
SUBROUTINE sub(i)
INTEGER*4 i
```

guarantees that the calling and called statement arguments will be compatible.

Unformatted Sequential I/O for Version 2.22 and Earlier

Command line or F77L.FIG /[N]U

When /U is specified, F77L creates unformatted sequential files that are compatible with programs generated by Versions 2.22 and earlier of F77L. The only reason for using this option is if you need to create an input file to a program already generated using F77L Versions 2.22 or earlier.

When /NU is specified, F77L will create unformatted sequential files that can be processed much faster.

WARNing Messages During Compilation

Command line or F77L.FIG /[N]W

If the /W WARN Option is ON, compiler warning messages are displayed. Experience indicates that warning messages should be resolved before continuing with linking and execution. When a WARNing is heeded, and the source is modified and recompiled, further warnings may be displayed. By compiling, modifying, recompiling and so on, the program is improved.

Specifying /NW suppresses most warning messages. Those not suppressed may cause FATAL or undetected errors in other FORTRAN implementations.

Cross-reference Listing

Command line or F77L.FIG /[N]X(spec=sval[,spec=sval]...)

Where:

> "spec" is "W" for page width, "L" for page length or "I" for listing INCLUDE files. The specifiers may be in any order.
>
> For "W=sval", the page width specifier, "sval" is an unsigned decimal constant. The minimum page width is 60. If not specified, the page width is 128. The headings are designed to work on a 128 character line, so some of them may not be as easy to read on a shorter line.
>
> For "L=sval", the page length specifier, "sval" is an unsigned decimal constant. The minimum page length is 20. If not specified, the page length is 57. The page length refers to the maximum number of lines F77L will print on a page. The printer must be adjusted to the paper page length.
>
> For "I=sval", the INCLUDE listing specifier, "sval" is either "Y" to list INCLUDE files or "N" to not list INCLUDE files. If not specified, INCLUDE files will be listed.

Specify the /X Option to cause F77L to generate a Cross-reference Listing. Specifying a listing file will not create a listing file unless either the /X or /H Option is specified. For details on the Cross-reference Listing, see Sections A.7 – A.7.5.

To send the hardcopy output directly to the printer, use the DOS reserved device names "LPT1", "LPT2" or "PRN" to specify the printer on your F77L Command Line. Specifying "CON" is not recommended. Examples:

The command line:

> **F77L MYPROG.F,,PRN**

causes F77L to send Cross-reference Listing output to the system printer.

On the other hand, the command line:

F77L MYPROG.F /X(W=100,L=50,I=N)

causes F77L to send Cross-reference Listing output to a file named "MYPROG.LST". The Cross-reference Listing output will have a width of 100, a length of 50 and INCLUDE files will not be listed.

Perform Production Code Optimizations

Command line or F77L.FIG /[N]Z1

Specifying this option causes the compiler to perform production code optimizations that may not be appropriate during debugging.

Specify /NZ1 to ensure the reliability of variables dumped in SOLD.

NOTE: These optimizations limit SOLD's and the P77L Profiler's ability to display variables. SOLD issues warnings for unreliable values.

A.7 Source/Cross-reference Listing

An F77L Cross-reference Listing is divided into several sections:

1. a listing of the source file with error messages;

2. a table of the line numbers of the executable code and the offset into the code segment of the code for the line number;

3. a description of the data storage areas used by the program unit;

4. a cross-reference listing of all symbols used; and

5. a cross-reference listing of all labels used.

To generate a source listing specify /H (see Section A.6.1) To generate all other listings specify /X (see Sections A.7 – A.7.5). The following describes the format of each section.

A.7.1 The Source Listing

The main listing heading tells the date and time that the compilation was performed, the compiler version, the program unit name and the compilation options used. The listing lines consist of three fields: the first (which is blank unless an INCLUDE file is being listed) will contain the letter "I" followed by the number of the INCLUDE file; the second is the number of the line in the file; and the third is the text of the statement. A tab at the beginning of the line tabs to column 7 if the character following the tab is blank or a letter, or to column 6 (continuation) otherwise.

If there are any errors detected in the first pass through the source, they are displayed below the line containing the error with a "^" (caret) pointing to the position of the error. Errors detected later will be printed after the END statement.

A.7.2 Line Number Table

The Line-number Table contains one entry for each executable statement in a program unit. Each entry consists of the line, followed by a colon, then the offset into the code segment of the machine language code that performs the executable statement. Since all executable statements are assumed to be in the same source file, no INCLUDE file numbers are displayed.

NOTE: If neither /S nor /L is specified for the compilation, this section will not be generated.

A.7.3 Data Storage Area Map

The data storage area map consists of 4 fields:

1. number of the storage area (referred to in **Symbol Cross-Reference Listing**);

2. name of storage area (if applicable) or type of storage (Blank COMMON, Local Stack and SAVEd Local Data);

3. external segment name of area origin; and

4. number of bytes in the area in hexadecimal.

A.7.4 Symbol Cross-reference Listing

The Symbol Cross-reference Listing consists of six fields:

1. the name of the symbol;

2. the type of the symbol (if applicable);

3. the class of the symbol;

4. number of the storage area where allocated;

5. the offset into the storage area. Data allocated on the Local Stack will be of the form: BP+FFFE. SS:[BP] points to the end of the subprogram's local data (BP and SS are 80x86 registers); and

6. a list of the references of the symbol.

The reference list items consist of the line number of the reference, followed by the INCLUDE file number if applicable, followed by a character to indicate the type of the reference.

The meaning of the characters is as follows:

- **a - label assigned to symbol in ASSIGN;**
- **g - assigned symbol used in GO TO.**
- **f - symbol used as a format;**
- **d - symbol used as a DO index;**
- **r - symbol is passed as an argument.**
- **= - symbol assigned a value;**
- **u - the symbol is used in an expression;**
- **i - the value of the symbol is input;**
- **o - the value of the symbol is output;**
- **/ - the value of the symbol is initialized as in a DATA statement;**
- **s - the symbol is specified in a type, COMMON or EQUIVALENCE statement.**

A.7.5 Label Cross-reference Listing

The Label Cross-reference Listing consists of two fields:

1. the label number; and

2. a list of the references of the label.

The reference list items consist of the line number of the reference, followed by the INCLUDE file number if applicable, followed by a character to indicate the type of the reference. The meaning of the characters is as follows:

- **a - label assigned to symbol in ASSIGN;**
- **i - label used in arithmetic IF;**
- **g - label used in GO TO.**
- **@ - label appears on a FORMAT statement;**
- **f - label used as a format;**
- **d - label used as a DO terminator;**
- **r - label is passed as an argument.**
- **s - labelled statement.**

NOTE: If no labels are used in the program unit, this section will not be generated.

A.7.6 Configuration Suggestions

The configuration file "F77L.FIG", a one-line ASCII file, is delivered as follows:

/N0/N2/N3/N7/NA/NA1/NB/NC/ND/NE/NF/NH/NI/NK/NL/O/P/R/S/NT/NU/W/NX/NZ1

which sets the following compiler options:

use Standard implicit typing rules (/N0);

don't generate smaller and faster code for the 80286 or 80386 (/N2);

don't generate 80386 code (/N3);

allow adjustable dimension arrays greater than 64K bytes (/NA);

arrays dimensioned (1) are assumed to be less than 2**20 bytes long (/NA1);

suppress the output of warning messages indicating nonconformances to the FORTRAN 77 Standard (/NC);

create F77L-generated direct files with headers (/ND);

require an NDP coprocessor (/NE);

accept standard-format source files (/NF);

don't generate Weitek 1167 or 3167 code (/NK);

output compiler options and warning messages (/O/W);

don't produce hardcopy and cross-reference output (/NH/NX);

produce object modules that have the least overhead (/N7/NB/NI/NL), while protecting numeric constants (/P);

allocate local subprogram variables and arrays so their values are available the next time the subprogram is executed (/R);

generate a ".SLD" file for SOLD (/S).

set default lengths for INTEGER and LOGICAL types to 4 bytes (/NT);

perform faster unformatted sequential I/O (/NU); and

perform no optimizations that result in limiting the SOLD and P77L Profiler interfaces (/NZ1).

You can customize your F77L Compiler by editing "F77L.FIG" with "FIG.EXE" or any ASCII editor to set the options the way you want them. For example, to have the compiler generate code to check subprogram interfaces and to generate a line-number table and hardcopy and cross-reference output (all are recommended), change the line:

/N0/N2/N3/N7/NA/NA1/NB/NC/ND/NE/NF/NH/NI/NK/NL/O/P/R/S/NT/NU/W/NX/NZ1

to

/N0/N2/N3/N7/NA/NA1/NB/NC/ND/NE/NF/H/I/NK/L/O/P/R/S/NT/NU/W/X/NZ1

Try compiling "DEMO.FOR" using different options. You can't hurt anything, and it might smooth the learning curve; for instance, try:

F77L DEMO /N0
F77L DEMO /N0/O
F77L DEMO.FOR /P/L/I

These few compilations illustrate how you can change compiler options and how they are displayed.

If you are familiar with the FORTRAN program you are working with, then you are the best judge of which options to use. If you are working with unknown code, we suggest the following options be utilized:

/N0/N2/N3/N7/NA/NA1/B/NC/D/NE/H/I/NK/L/P/R/S/NT/NU/W/X/NZ1

which will protect you from many problems. It is our users' experience that many "production programs" have bugs that are detected by these options.

NOTE: If your program is specifically targeted for an 80386 machine and will never have to run on an 8086, 8088, or 80286 machine, use the /3 Option to generate better code. If your program will never have to run on an 8086 or 8088 machine, then use the /2 Option to generate better code.

Many users create a subdirectory for each project they are working on. An "F77L.FIG" File could be placed in each subdirectory with the compiler options set appropriately for that project.

A.8 Converting Source File Formats

The program "STDTOFF.EXE" converts standard-format source files (card image) to free-format source files. The program "FFTOSTD.EXE" converts free-format source files to standard-format source files. The command lines for running these programs are:

STDTOFF standard.ext freeformat.ext

or

FFTOSTD freeformat.ext standard.ext

Where:

"STANDARD.EXT" is a standard-format source file,

and

"FREEFORMAT.EXT" is a free-format source file.

A.9 Customer Support

Our Technical Support Staff is available from 9:30 a.m. to 3:30 p.m. (PST) Monday through Friday to answer your questions. If you find a problem in your program which appears to be caused by the F77L Compiler or runtime system, please send us as small an example as possible. Send the example on a disk in a disk mailer, or via Telex (9102401256), FAX (702 831-8123) or the Lahey BBS (702 831-8023). Include your compiler disk serial number, compiler version number, configuration file ("F77L.FIG") date and size with a short description of the problem. We will get back to you with a correction if it is our problem, or an explanation if it is not.

You can patch the F77L Compiler at your site by modifying the ASCII file "F77L.FIX". The compiler reads "F77L.FIX" before beginning a compilation and modifies its execution accordingly. The patches might be mailed, dictated over the phone, sent by FAX or posted on our Electronic Bulletin Board System in response to a reported problem.

Patches can be added at the end of "F77L.FIX" with any ASCII editor that does not add any extra control characters except carriage return, line feed and end-of-file.

A semicolon and any text in a patch which follows is a comment and need not be entered. Be sure to enter the patch text exactly as you receive it; in particular, note the difference between the letter "O" and the number "0".

You can use the Lahey Bulletin Board System (BBS) to perform a variety of tasks. Its most important feature is the quick processing of problems and patches. With the BBS you can upload programs with problems in them and then download a new "F77L.FIX" File to solve the problem. The BBS telephone number is **(702) 831-8023**.

RESERVED FOR YOUR NOTES

B PROGRAMMING HINTS

B.1 Efficiency Considerations

In the majority of cases, the most efficient solution to a programming problem is one that is straightforward and natural. It is seldom worth sacrificing clarity or elegance to make a program more efficient. If a fully debugged program is found to be too slow or to use too much input/output, it is usually better to improve the algorithm than to resort to trickery to "get better code". Language-independent optimizations (such as moving invariant computations out of loops) are always effective.

The following observations, which may not apply to other implementations, should be considered in cases where program efficiency is critical:

1. Start each array dimension at zero (not at one, which is the default). Thus, declare an array "A" to be "A(0:99)", not "A(100)".

2. One-dimensional arrays are more efficient than two, two are more efficient than three, etc.

3. The "/NI" (INTERFACE), "/NB" (BOUNDS), "/NL" (LINE) and "/Z1" (PRODUCTION CODE OPTIMIZATIONS) Compiler Options may be used to optimize a program that is fully debugged. These options generate the smallest and fastest executing program possible, but do so at the expense of runtime diagnostics. These options should only be used with good reason and considerable restraint.

   BOUNDS checking causes more code to be executed for each reference to an array with a nonconstant subscript. The extra code size and number of

instructions executed varies according to the size of the array and the number of array references in a statement.

INTERFACE checking requires four instructions (19 bytes) to invoke a function and seven instructions (31 bytes) to invoke a subroutine. It also requires six instructions (20 bytes) at the beginning of a function and nine instructions (30 bytes) at the beginning of a subroutine.

LINE numbers do not increase execution time, but do use 4 bytes of memory for each executable statement in a program unit.

4. Compile as many subprogram source files as practical together. The actual files may remain separate by creating a file of INCLUDE statements and compiling that one file. For example, the file "SUBS.FOR" could contain the following:

```
INCLUDE  SUB1.FOR
INCLUDE  SUB2.FOR
INCLUDE  SUB3.FOR
   .        .
   .        .
   .        .
```

   Compiling SUBS.FOR allows you to compile each of the INCLUDEd files.

5. Make a direct file record length a power of two.

6. Unformatted input/output is faster for numbers; direct access is usually the fastest.

7. Formatted CHARACTER input/output is faster using:

```
CHARACTER*256 C
```

   than:

```
CHARACTER*1 C(256)
```

B.2 NDP Information

NDP Error ...

This runtime error message indicates one of a variety of error conditions diagnosed by the Numeric Data Processor (NDP). A FORTRAN program generates this message with the following errors:

Using a REAL variable containing invalid data (from not being initialized, from being initialized with CHARACTER or hexadecimal data or from improper argument passing).

Attempting an assignment to an INTEGER or INTEGER*2 target of a value too large for the target (integer overflow).

Dividing zero by zero (all other divisions by zero generate the divide-by-zero error message).

System subroutines OVEFL or UNDFL were used, and a number has degenerated to an "unnormal" or "denormal".

The NDP was used by a non-F77L subprogram, and the state of the NDP after the call was different from its state before the call.

See INTEL or other documentation for more information about the NDP.

B.3 FORTRAN Side Effects

F77L arguments are passed to subprograms by address, and the subprograms reference those arguments as they are defined in the called subprogram. Because of the way arguments are passed, the following side effects can result:

Passing a CHARACTER argument to a subprogram and trying to process it as if it were a different data type (or vice versa) can cause random locations to be modified and unpredictable program behavior.

Declaring a dummy argument as a different numeric data type than in the calling program unit can cause unpredictable results and NDP error aborts.

Declaring a dummy argument to be larger in the called program unit than in the calling program unit can result in other variables and program code being modified and unpredictable behavior.

If a variable appears twice as an argument in a single CALL statement, the corresponding dummy arguments in the subprogram will refer to the same location. Whenever one of those dummy arguments is modified, so is the other.

If the "/NP" Compiler Option is used for compiling a program unit, constants passed to subprograms are not protected from being modified. Passing the constant "5" as an argument to a subprogram and modifying the dummy argument corresponding to it in the subprogram can result in subsequent references to the constant "5" being incorrect in all program units compiled with the calling program units.

Function arguments are passed in the same manner as subroutine arguments, so that modifying any dummy argument in a function will also modify the corresponding argument in the function invocation. For example:

```
Y = X + F(X)
```

The result of the preceding statement is undefined if the function "F" modifies the dummy argument "X".

B.4 Recursive Programs

F77L subprogram code is reentrant so that subprograms can be invoked recursively. Recursion occurs when a subprogram invokes itself, either directly, or indirectly through other subprograms. Unintentional recursive invocation is not diagnosed by F77L and will probably result in the program aborting by running out of stack.

In order to have a recursive subprogram work correctly, it should have a separate copy of each local variable for each invocation of the subprogram which is currently active. To accomplish this, compile the subprogram with the "/NR" Compiler Option in either the "F77L.FIG" File or on the command line.

NOTE: Use of the "RECURSIVE" keyword deactivates the "/R" (REMEMBER) Option for the recursive program unit so that local variable space will be allocated for each invocation of the function or subprogram. Variables appearing in a SAVE statement or in a DATA initialization context will be statically allocated, so that values will be retained across function and subroutine invocations.

B.5 F77L File Formats

B.5.1 Formatted Sequential File Format

Formatted sequential records consist of a sequence of ASCII characters followed by a carriage return and a line feed. A control-Z as an end-of-file indicator is optional for input files, but is not written by the F77L input/output system.

B.5.2 Unformatted Sequential File Format

Unformatted sequential files begin with a 1-byte header with the value hex "FD". A record-length indicator appears at the beginning and end of each record.

The format of the record-length indicator at the beginning of a record is as follows:

> The lower two bits of the of the first byte indicate the number of additional bytes (0–3) that will follow. The upper 6 bits of the first byte are low-order bits of the record length. The following byte (if indicated) contains the next 8 bits of record length, and so forth. This record length does NOT include the length of the record-length indicators.

The format of the record-length indicator at the end of the record (which is processed backwards by the BACKSPACE statement) is as follows:

> The lower 2 bits of the last byte indicate the number of additional bytes (0–3) that will precede it. The upper 6 bits of the last byte are the low-order bits of the record length. The preceding byte (if indicated) contains the next 8 bits of record length, and so forth. This record length includes the length of the record-length indicator at the beginning of the record.

NOTE: The record-length indicator at the beginning of a record may be larger than required, i.e. it may contain high-order zero bytes. The record-length indicator at the end of the record will never be larger than required.

B.5.3 Direct File Format

The first record (record number 0) of a direct file contains system information. It is the same size as all other records to facilitate increased input/output efficiency. The first byte of record 0 is a header in which the high-order 6 bits designate unformatted direct (hex F4) or formatted direct (hex F8), and the low-order 2 bits indicate the range of the record length. If the value in those two bits is 1 or 2, then the value is the record length. If the value is 3, then the record length is stored in the second and third bytes of the file (low-order byte first).

B.5.4 Transparent File Format

Files opened with ACCESS="TRANSPARENT" are processed as a stream of bytes with no record separators. While any format of file can be processed transparently, the programmer must know its format to process it correctly.

B.6 INTEGER*2 Arithmetic

Caution must be exercised when using INTEGER*2 arithmetic because overflow checking is not done for most operations. For example, in the following program fragment, the answer is incorrect because 1000 * 1000 is too big for an INTEGER*2 target.

NOTE: The result of an INTEGER*2 operation must be less than 32767 or the result of the operation will be incorrect, even if the target variable is INTEGER*4.

```
INTEGER*2 I,J,K
DATA I/1000/, J/1000/
K = I*J
END
```

RESERVED FOR YOUR NOTES

Not All Lahey Language Systems Include All of These Tools.

Not All Lahey Language Systems Include All of These Tools.

C LAHEY BLACKBEARD EDITOR

The Lahey Blackbeard Editor is a full-featured FORTRAN program development tool that supports compiling, linking, and making, all from within the editor's windows. It also includes a special Lahey FORTRAN Language System Expansion Window that provides templates for all Lahey FORTRAN statement constructs which can be pasted into users' source code to save development time. The editor includes mouse support although one is not required to use its windowing capabilities. Another feature allows users to define the editor's function key bindings to match those used by other popular editors. The editor can be easily swapped out of memory to give developers all but 1KB of their RAM back for compilation or program execution.

Documentation for all functions is provided in the form of menu and help systems. The help system can be printed using utilities described below. This appendix will help you begin using Lahey Blackbeard and show you how to use the help and menu systems and some features unique to Lahey FORTRAN.

C.1 Getting Started

To begin using Lahey Blackbeard, change to the \F77L directory and type:

LB

At the "filename:" prompt type:

SAMPLE (or any filename)

If you typed the name of a new file, Lahey Blackbeard displays the message:

File does not exist. Create?

Highlight "Yes" and press **ENTER**.

Lahey Blackbeard then displays edit window number 1 (as indicated by the "1" in the lower-left corner of the screen).

Press the **ESC** Key to display Lahey Blackbeard's Main Menu. Be sure to remember the keys assigned to the "Help", "Abort", and "Exit" Functions for later use.

Highlight the "Help" Option on the Main Menu and press **ENTER** to display the first help screen. Read the screen and note the three introductory help topics:

> **Running Lahey Blackbeard**
> **System Configuration**
> **Menus**

Use the **Up Arrow** and **Down Arrow** Keys to highlight one of these help topics and press **ENTER** to display help on the selected topic.

NOTE: The options are underlined instead of highlighted on monochrome monitors.

Follow the instructions on each topic:

> **Running Lahey Blackbeard** — shows how to invoke Lahey Blackbeard with various command-line options.
>
> **System Configuration** — shows how to set the DOS environment variables. Refer to **Appendix A — Lahey Blackbeard Variables** for recommended environment variable settings.
>
> **Menus** — describes the menu system and how to use it to learn the Lahey Blackbeard Commands.

C.2 Lahey Blackbeard Features

C.2.1 Lahey Blackbeard Function Keys

The Lahey Blackbeard function keys mentioned in this appendix are referred to by function name and not actual key names since Lahey Blackbeard's function keys are user configurable. The Key Binding Feature allows you to assign any Lahey Blackbeard function to almost any key. From Window 1, do the following to display a list of all current key assignments (key bindings):

1. Press **ESC** to display the Main Menu.
2. Highlight "Options/Settings/Keys" and press **ENTER**.
3. Highlight "Show Bindings" and press **ENTER**. A list of all current key assignments displays.
4. After viewing the list, press **ESC** to return to Window 1.

C.2.2 Selecting Lahey Blackbeard Windows

Lahey Blackbeard is a windowed editor with 16 windows: ten for editing and six for special purposes. In windows 1 through 10, you can edit one or more files. The six special windows allow you to:

Compile files and Review error messages (Compile and Review Windows)
Enter DOS commands (DOS Window)
Edit the Paste Buffer (Paste Window)
Edit the Language Expansion File (Language Window)
Edit text you have deleted (Kill Window)

To switch to a different window, press the **Select** Key to display a list of all windows, highlight the window you want, and press **ENTER**. To switch to a different window and load a file, press the **Switch to Window** Key. You will be prompted for the window number and the name of the file. To go directly to a window, press, for example, the **Window 1** Key. All windows, including the six special windows, can be bound to a key. Press the **Show Bindings** Key to view the window key bindings.

C.2.3 Language Expansion Feature

The Lahey Blackbeard Language Expansion facility allows substitution of language keywords by pre-defined language constructs. For example, if you type:

OPEN

and press the **Expand LNG** Key, the following construct displays:

OPEN (UNIT=, openspec=sval)

Move the cursor to the comma following "UNIT=" (by pressing the **Word right** Key twice) and type the unit number "1":

OPEN (UNIT=1, openspec=sval)

Move the cursor to the expansion symbol, "openspec=sval", and press the **Expand LNG** Key again to display a list of OPEN specifiers and specifier values.

```
Language Construct Expansion

FILE=, openspec=sval

FORM='FORMATTED', openspec=sval
FORM='UNFORMATTED', openspec=sval
FORM=, openspec=sval

STATUS='NEW', openspec=sval
STATUS='OLD', openspec=sval
.
.
.
```

Highlight "FILE=, openspec=sval" in this list, and press **ENTER** to display the following:

OPEN (UNIT=1, FILE=, openspec=sval)

Continue to move the cursor through the OPEN statement, expanding the "openspec=sval" symbol and selecting a specifier. When you select specifiers that require arguments not supplied (e.g. "FILE=,"), advance the cursor to the comma and type the argument. When you finish building the OPEN statement, use the **Delete-to-EOL** key to remove the remaining expansion symbol ("openspec=sval"), and then type the closing, right parenthesis.

NOTE: For example, to display FORTRAN Language constructs for the OPEN statement, you only have to type the unique characters, i.e. **OPE** and press the **Expand LNG** Key. See section 9.1 for a list of FORTRAN Input/Output keywords.

Though you can create your own language expansion file, we provide the FORTRAN-Language expansion file, "LB.LNG". If you "SET" the environment variable, "LBLNG", equal to "d:\F77L\LB.LNG, this file is automatically loaded and will perform as described when Lahey Blackbeard is invoked.

C.2.4 Invoking F77L from within Lahey Blackbeard

Lahey Blackbeard offers the ability to compile, link, and make while editing a file. From Window 1, press the **Compile Settings** Key to display the "Compile Settings" window:

Function	Execute String	Review File	Review Window	Swap Out
COMPILE	F77L%f>compile.msg	compile.msg	Review Window	Yes

As shown above, we provide an "Execute String" to invoke Lahey F77L from within Lahey Blackbeard. Return to Window 1 by pressing **ESC.** For example: From Window 1, press the **New File** Key. Then type the full pathname of the "ERRORS.FOR" File and press **ENTER** to display "ERRORS.FOR" in Window 1. The "ERRORS.FOR" File is a sample file included on the F77L distribution disks that contains intentional programming errors.

To compile "ERRORS.FOR", press the **Compile file** Key. Two windows display: the "Compile Window" and the "Review Window." By pressing the **Next error** and **Previous error** Keys, you can move through the error messages in the "Review Window" while the cursor automatically moves to the line containing the error in the "Compile Window." Press the **Compile settings** Key, then the **Help** Key to learn more about "Execute Strings."

C.2.5 Printing the Help System

To print the help system to a file, go to the \F77L directory and type:

LBMAN

This generates the "LB.DOC" File containing the entire Lahey Blackbeard help system. You can now browse the file with the editor itself by typing:

LB LB.DOC

To send a copy of the file to a printer type:

TYPE LB.DOC > PRN

C.2.6 Lahey Blackbeard Mouse Support

Lahey Blackbeard supports the Microsoft Mouse Driver. The mouse driver must already be loaded. In general, the left mouse button functions as the **ENTER** Key and the right mouse button functions as the **ESC** Key. The right mouse button selects menus and also escapes from a menu without selecting an item and returns to the Main Menu from a sub-menu. The left mouse button selects items from a menu when the cursor is positioned on an item.

D LAHEY MAKE BY OPUS SOFTWARE

Lahey MAKE is an implementation of the UNIX program maintenance utility called MAKE. It has the same syntax and all of the capability, as well as some DOS specific enhancements. Lahey MKMF (make makefile) is a tool for generating and managing make-files.

Throughout this appendix, the symbol "SPACE" is a single space character and the symbol "WHITESPACE" is one or more spaces or tab characters.

D.1 MAKE Utility Description

MAKE is a utility that helps you maintain programs, particularly programs that are composed of several modules (files). MAKE has several features that make it very useful for software development:

1. **Module Version Tracking** — Once you describe the relationships between modules, MAKE will keep track of them and only "make" those that are out of date with respect to their sources. You don't waste time figuring out which files you've changed, which have to be recompiled, how to link them, etc. Most importantly, you'll never forget to compile a file. MAKE uses the file time to determine which files are newer than others making it imperative that you either have a real-time clock or are diligent about setting the time and date when you boot up.

2. **Reduced Memory Usage** — Two options can be used to reduce the amount of memory MAKE uses to as little as a few hundred bytes. This is ideal for memory-cramped systems.

3. **Automatic Response Files for OPTLINK and Lahey Library Manager** — Under DOS, commands must be shorter than the command line limit of 128 characters. Since OPTLINK is used often when doing modular programming, MAKE knows about it specifically and will automatically generate a response file if the OPTLINK Command is longer than the limit. MAKE will do a similar thing with the Lahey Library Manager.

4. **Multiple Directory Support** — Often when doing program development, you want to create several different versions of a program, but keep only one set of sources. Using the VPATH feature of MAKE, it is possible to have sources in directories other than the current directory and have MAKE find them (see section D.11).

5. **Conditional, Flow and Other Directives** — These make it possible to read a makefile conditionally (if/elif/else), to iterate on shell lines (foreach/while) and to include other makefiles. The conditional forms include several types of comparison and logical operators in a C syntax. These forms also allow the testing of the state of files (e.g. existence, writability, length, etc.). See section D.7.1 for details.

6. **Better I/O Redirection than DOS** — MAKE can redirect all messages produced by a compiler into a file. This file could then be processed by an editor which automatically positions you at the errors. See section D.12 for details.

7. **Additional MAKE features:**

   Maximum Work Flag — A command-line option to force MAKE to ignore errors, but not make a target that depends on another target that was in error. See the –k Flag in section D.10 for details.

Directory Snapshot Option — A command-line option to increase processing speed by causing MAKE to read entire directories at startup time and never return to the disk for file information. See the –D Flag in section D.10.

Microsoft Makefile Compatibility Flag — This flag allows MAKE to read and act on Microsoft makefiles. See the –M Flag in section D.10.

D.2 What is MKMF?

MKMF ("make makefile" or "McMuff") automates the process of creating and maintaining the makefiles that MAKE uses. MKMF's features include:

1. creation of makefiles from user-defined templates.

2. automatic update of several important macros (OBJS, SRCS, etc.).

3. automatic dependency generation. MKMF analyses the source files in user-specified directories, checks for include files, and builds a dependency list for each corresponding object file. Each include file is also scanned and the resulting dependency information (which files depend on which others) is appended to the makefile. If desired, MKMF can fully understand C preprocessor directives when searching for include files. See the –c Flag in section D.20.

D.3 MAKE Functions

MAKE executes commands in a MAKE description file to update one or more targets. The targets are typically the names of programs or files.

MAKE updates a target if it is older than the files it depends on, or if the target does not exist. If no targets are given on the command line, the first target in the makefile is "made". MAKE [options] are discussed in section D.10 and [macros] are discussed in section D.6.

D.3.1 MAKE Command-line Syntax

The MAKE command-line syntax is shown below:

make [options] [macros] [targets]

to use the small-data model MAKE (64KB of data), or:

makel [options] [macros] [targets]

to use the large-data model (1MB of data) if the small data model runs out of memory:

D.4 Getting Started

The general procedure for using a "make" program is:

1. Create a subdirectory for each project you are working on.

2. Put only project source files in this subdirectory. Do not keep any other source files in here.

3. Change directory (chdir) to that subdirectory.

4. At the DOS prompt, type:

   mkmf -dL -l

Answer the two questions. The first answer is the name of the program you are working on (example "TEST.EXE"), the second answer is the name of the directory you want "TEST.EXE" to be put in when you ask MAKE to "install" the program. Hitting **ENTER** to each question will accept the displayed default.

This will create a file called "MAKEFILE" which is the file the MAKE program reads. If a "MAKEFILE" does not get created, MKMF did not find the "MAKEFILE.PRG" makefile.

Check to make sure you put "MAKEFILE.PRG" in the directory named by the INIT environment variable.

5. Next, at the DOS prompt type:

   make -n

   You should see a list of commands that will be executed to compile your project source files and link them into your project executable file. The "-n" tells MAKE to show what it will do, but not to actually do it.

   If you are satisfied with these commands, type:

   make

   Now your compiles, assemblies, and links should occur.

6. You can edit and modify MAKEFILE to change the values of compiler flags, libraries to be linked in, and other macros. If you change the names of the sources, or if you add more source files, do an "mkmf" again. This will recompute some of the macros in MAKEFILE.

7. Refer to the MAKE and MKMF documentation if this does not work as expected.

D.5 MAKE Description Files

MAKE uses two description files: "MAKE.INI" and "MAKEFILE". The description files consist of several entries:

1. dependency and shell lines (section D.5.2)

2. macro definitions (section D.6)

3. conditional and loop directives (section D.7.1 and D.7.3)

4. inference rules (section D.9)

5. pseudotargets (section D.8)

When MAKE starts up, it looks for an initialization file called "MAKE.INI". This file usually contains only inference rules and macro definitions that you don't want to put in every makefile, and is generally used to customize your copy of MAKE. The initialization file is searched for first in the current directory, then in the directory specified with the "INIT" environment variable.

D.5.1 Description File Details

Lines in a MAKE description file that are too long to fit on one line can be extended to the next line by putting **ENTER** as the last two characters of the line. If you need "\" as the last character on the line, use either **ENTER** or **WHITESPACE ENTER**. There is no limit to total line length (except available memory).

The comment character is "#". Everything on a makefile line including and after the comment character is ignored. If you need to have a comment character in your makefile, use "\#". Continuation of a line is checked before the existence of the comment character, so continued lines are affected by a comment character on the continuing line.

D.5.2 Dependency and Shell Lines

These are lines that specify the relationship between targets and their prerequisites, and how to update the targets. The general form is:

```
target [target...] :WHITESPACE[prerequisite...]
[WHITESPACEcommand]
```

The first line of an entry is a blank-separated list of targets, then a colon, then a list of prerequisite files. This is known as a dependency line. All following lines that begin with WHITESPACE are commands to be executed to update the target. These lines are also known as shell lines since they are executed in a subshell outside of MAKE.

As an example, assume you have a program "TEST.EXE" that is composed of modules "MAIN.OBJ" and "SUB.OBJ". These ".OBJ" files are compiled from their respective ".FOR" FORTRAN source files. The makefile might look like:

```
test.exe : main.obj sub.obj
  optlink main.obj sub.obj, test.exe;

main.obj : main.for
  f77l main.for

sub.obj : sub.for
  f77l sub.for
```

A target may appear on the left of more than one dependency line, and it will depend on all of the names on the right of the colon on those lines, but shell lines may be specified at only one place in the make file.

NOTES: After the shell lines to update a target have been executed without error, MAKE assumes the target is up to date. If you give commands that do not really update a target, MAKE does not know.

The character separating targets and prerequisites, ":", is also the character used for the drive separator in DOS. To distinguish this colon from the drive separator, it must be followed by space or tab (target:WHITESPACE), semicolon (target:;), colon (target::), or nothing (target:ENTER). If you consistently use at least one space you will have no problem.

Targets defined in the makefile replace targets of the same name defined in "MAKE.INI".

As with UNIX make, dependency lines can have a shell line on the same line by following the last prerequisite with a semicolon and the shell line. For example:

```
test.exe : test.obj; optlink test;
```

is equivalent to:

```
test.exe : test.obj
  optlink test;
```

D.5.3 Just Type "Make"

By default, if no targets are specified on the command line, the first target in the makefile gets made. This is the UNIX Make compatible behavior and it differs from Microsoft Make, which processes each target in order of appearance in the makefile. The –M Option results in behavior compatible with Microsoft Make (see section D.10 for details). On the other hand, Microsoft NMAKE behaves like Lahey MAKE and the –M flag is not required.

A target with no prerequisites is considered up to date (already made) if it exists and out of date (must be made) if it doesn't exist. A dummy target (a file that never exists so is always out of date) is often used as the first target in the makefile when you wish to make several targets without specifying each name on the command line. For example, consider this makefile:

```
all : target1.exe target2.exe
target1.exe :
  commands
target2.exe :
  commands
```

Running MAKE with no command-line arguments results in both "TARGET1.EXE" and "TARGET2.EXE" being made. This happens because MAKE always ensures all prerequisites are up to date before it makes a target. Here, the target "all" has two prerequisite files: "TARGET1.EXE" and "TARGET2.EXE". First "TARGET1.EXE" then "TARGET2.EXE" are made, then MAKE checks if either of these files are more current than "all". Since "all" does not exist, both are newer than it, and the shell lines to update "all" will be executed. Here, there are no shell lines, so nothing further gets done. For this reason, with the proper makefile setup you can "just type make".

D.5.4 Double Colon Dependency Lines

If a dependency line appears with a double colon, "::", separating targets and prerequisites, then the command sequence following that line is performed only if the name is out of date with respect to the names on the right of the double colon, and is not affected by other double colon lines on which that name may appear. Consider the following makefile:

```
1:: 2; echo 2
1:: 3; echo 3
```

If 2 and 3 are more recent than 1, then "make –n 1" will result in:

```
echo 2
echo 3
```

If only 2 is more recent than 1, the response is:

```
echo 2
```

If 1 is more recent than both 2 and 3, the response will be:

```
Lahey MAKE: '1' is up to date.
```

D.5.5 Shell Line Execution

Under DOS, when MAKE executes shell lines, such as "MASM", it checks if it is an DOS internal command, if it's a batch file, or if it involves piping. If so, MAKE spawns the shell named in the "COMSPEC" environment variable and passes it the command line. Otherwise, MAKE tries to execute the command as a program (an ".EXE" or ".COM" file). This is done to prevent loading the secondary shell unnecessarily.

D.5.6 Shell Line Prefixes: –, @, =, ^

Shell lines may be prefixed with special characters which modify MAKE on a shell line by shell line basis. The prefixes appear as the first character(s) on a shell line and several may be used on a given line. Only "–" and "@" are compatible with UNIX make.

@ Causes the shell line to not be printed before execution.

— Means ignore the exit status (a number testable by "if errorlevel") that is returned from the shell line. Normally MAKE stops when a shell line returns a nonzero exit status (see section D.14). Additionally, the "status" macro gets set to the exit status of the shell line only for lines with the "–" prefix.

= Normally, MAKE will only look on the disk for a particular file once. Thereafter MAKE keeps track of it internally. An "=" before any of the target's shell lines causes MAKE to re-search the disk for the target after all shell lines have been processed. This can be used when the shell lines either do not create the target or create it in a different VPATH directory, or do not create it with the current system time.

^ This causes the shell line to be executed in 1-command mode. Normally, when a shell line is executed from inside of MAKE all of MAKE remains in memory and the shell line has less memory to work with. While the shell line is being executed, MAKE does nothing. When the shell line is finished, MAKE regains control. When a shell line is prefixed with "^", that shell line overlays MAKE and MAKE does not regain control.

D.5.7 Multiple Command Shell Lines

Under DOS, MAKE supports multiple commands in a single shell line if the shell line is enclosed in parentheses, "( )", and the commands are separated by semicolons (use \; if you need a semicolon to be interpreted literally). This is compatible with the UNIX shell. For OS/2 compatibility, rather than using ";" and parentheses, instead use " & " (i.e. SPACE&SPACE) between commands.

In fact, since each shell line is executed in its own shell, this is the only way to use DOS commands like "SET" and "CHDIR" so that they have any effect. For example, doing "make x" where the makefile looks like:

```
x:
  chdir \tmp
  copy *.* a:
```

will copy the contents of the current directory, not "\tmp", to Drive A. This is because MAKE starts each shell line in the current directory. The correct way to write this is:

```
x:
  (chdir \tmp; copy *.* a:)

     or

x:
  chdir \tmp & copy *.* a:
```

A batch file and "COMMAND.COM" are always used to execute multiple commands. Thus, you can force Lahey MAKE to use "COMMAND.COM" for any command by putting the command in parentheses. One interesting use of the multiple-command syntax is to create a single macro that expands to multiple commands.

D.5.8 Echo

Under DOS, the ECHO Command has been built into MAKE so there is no need to load a secondary command processor just to output a message to the screen. If you need the DOS version of echo, use parentheses (i.e. (echo ...)).

D.6 Macros

Makefile entries of the form:

```
name = [value]
```

are macro definition lines. Macros allow the association of a name and a value. Subsequent appearances of either $(name) or ${name} are replaced by value. If the name is a single character, the parentheses or braces are optional. Any whitespace between name and =, and between = and value are ignored. If value not given, the macro value is a null string. The previous makefile example could have been written as:

```
OBJS = main.obj sub.obj

test.exe : $(OBJS)    # [KEY]
  optlink $(OBJS), test.exe;

main.obj : main.for      # Main.obj depends on main.for
  f77l main.for

sub.obj : sub.for        # and sub.obj depends on sub.for
  f77l sub.for
```

When reading the makefile, MAKE expands macros on dependency lines only, such as the line indicated above with the [KEY] symbol. Hence the "OBJS" Macro would have to be defined before that line. Conventionally, all makefile macro definitions appear at the top of the description file. Macros on macro definition lines and on shell lines are left unexpanded. A macro name may be recursively defined in terms of other macros. If $(FOO) evaluates to BAR and $(BAR) evaluates to BAZ, then $($(FOO)) is BAZ. The macro value may be defined

in terms of the value of other macros, provided the value does not circularly refer to itself. That is, the following would be an error:

```
macro1 = $(macro2)    # macro1 = the value of macro2
macro2 = $(macro1)    # macro2 = the value of macro1
```

D.6.1 Incremental Macros

The value of a macro can be incrementally set, combining the current value of the macro with an incremental value. For example, with the value of the FFLAGS macro being "/NO", the following:

```
FFLAGS +=/E
```

causes the value of FFLAGS to be "/NO/E". Omitting any whitespace between the "+=" and the incremental value pastes the incremental value right after the current value. Otherwise, one space is used as the separator between the current and incremental values. Currently, incremental macros do not work from the command line.

D.6.2 Macro Precedence

Macros can come from the environment, the command line, makefiles, or can be built into MAKE. The origin of the macro determines how high its precedence is. In order to redefine the value of an existing macro, the new assignment must have at least as high of precedence. The default precedence of macro definitions is:

1. built-ins (highest)
1. command line definition
2. makefile definition
3. environment definition (lowest)

If "MAKE.INI" has macros EM=E and FFLAGS=/NO/$(EM), but you do:

```
make EM=NE
```

then when it comes time to use FFLAGS, MAKE will expand FFLAGS as "/NO/NE". This is because command-line macros have a higher precedence than makefile macros. For command line macros that include spaces, place the macro value in double quotes:

```
make FFLAGS="/NO /NE"
```

By default, the lowest precedence for a macro is the environment definition. This refers to the environment variables put into the DOS environment with the "SET variable = value" DOS command. For example, you could refer to the ${PATH} macro without defining it in "MAKE.INI" or makefile because its value comes out of the DOS environment.

D.6.3 Predefined Macros

MAKE maintains several macros automatically. For the most part, you cannot change the value of any of the macros listed here:

$@ – the current target name.

$* – the current target name minus its extension.

$** – the complete list of prerequisites of a target.

$? – the list of prerequisites newer than the target.

$< – the complete name of the file that satisfied the inference rule. It is only valid within an inference rule (see section D.9).

$$ – a single dollar sign.

d – used to test whether a macro has been defined. It evaluates to 1 if the macro is defined and 0 if it is not. For example, $d(FOO) will evaluate to 1 if FOO is a defined macro and 0 if it is not defined. "d" takes one argument which is enclosed in either parentheses or braces.

CWD – the current working directory.

MAKE – the appearance of this macro on any shell line will override the –n Flag (no execute — see section D.10). By default, the value of this macro is the word "make", but it can be changed to whatever value is desired. The special property of overriding the –n Flags is always in effect. As an example, consider the makefile:

```
all:
  (cd small; ${MAKE} ${MFLAGS})
```

Doing "make –n" results in MAKE actually doing "(cd small; make –n)". That is, the –n Flag is ignored for shell lines that contain the MAKE macro. This is especially useful for debugging makefiles which invoke MAKE recursively.

MFLAGS – specifies the options MAKE will start with. Since macros can be found in the DOS environment, the following DOS command could have been used to tell MAKE to use silent mode:

```
set MFLAGS=–s
```

Furthermore, MFLAGS is filled in with all options supplied to MAKE so it can be used when invoking MAKE recursively to pass along the options, as in the above example.

MAKE_TMP – the name of the directory MAKE will use for temporary files (batch, response and paging files). If you wish, you can define MAKE_TMP in terms of other macros or environment variables. For example:

```
MAKE_TMP = $(TMP)
```

A RAM disk would be the best place for MAKE_TMP.

RETURN – the value of this macro is the RETURN character. One place this may be useful is to separate a single macro into multiple lines. See section D.6.4 for an example.

SPACE – the value of this macro is the SPACE character.

status – the status macro gets filled in with the return code of a shell line if the shell line is preceded with the ignore prefix, "–". You can examine the status with the conditional and loop directives described in section D.7.

SYSTEM – the value of the SYSTEM macro is MSDOS.

D.6.4 Macro Modifications

A macro may appear as:

$(name{, :}modifier[,modifier ...])

Where:

"modifier" can be one of the single letters B, D, E, F, R, or Z; the characters –, < or >; or an expression of the form "string1=string2". The initial separator between name and the first modifier may be either a colon or a comma.

Consider the value of macro "name" as a string of elements, each of which could be a complete pathname. Modifiers are applied in order of appearance to each element of the macro value.

D – specifies the directory part (the part of the pathname before the filename proper – ".\" if there is no directory part).

F – is the filename (everything after the last directory separator).

B – specifies the base name (the filename without an extension).

R – is the root of the filename (the pathname minus the extension).

E – is its extension (the null string if there isn't one).

Z – specifies the drive letter, terminated by a ":" (the current drive if there isn't a drive component).

–letter – chops the macro element off at the last occurrence of letter. If letter is not given, the element is shortened by one character.

> string – appends string to each macro element.

< string – prepends string to each macro element. The string is everything after the < or > up to the next comma.

string1=string2 – cause all occurrences of string1 in name to be replaced by string2. For substitutions, the characters "," and "=" are delimiters. If you need one of the delimiters in the substitution, use a backslash immediately before it (the backslash is known as a "quoting" character). Backslash backslash (\\) evaluates to a single backslash.

For example, with FILE=c:\src\main.for

```
$(FILE,D) =  c:\src\
$(FILE,F) =  main.for
$(FILE,B) =  main
$(FILE,R) =  c:\src\main
$(FILE,E) =  .for
$(FILE,Z) =  c:
```

The character modifiers can be combined for combinations of the components. For example $(FILE,FE) is "main.for.for" but $(FILE,F,E) is ".for" (start with $(FILE), take its filename, then take the extension).

With OBJS = 1.objSPACE2.obj the FORTRAN sources can be expressed as:

```
SRCS = $(OBJS,.obj=.for)
```

The OBJS macro can be broken up into several lines. $(OBJS,SPACE=SPACE+$(RETURN)) expands to:

```
1.obj +
2.obj
```

D.6.5 Make-time Macros

A command internal to MAKE, called "%set" (note the leading %), can be used to set the value of MAKE's macros at make-time. MAKE's environment also gets updated, so the environment of any shell spawned by MAKE also sees the new value. This internal command is used on a shell line just as you would use the DOS Set Command. Its main use is:

1. in .BEFORE, to set up the environment for the entire makefile;

2. in any shell line, to pass information to the ensuing shell lines, thereby overcoming DOS's 128 character command-line limit. One example of the second usage is for using the Microsoft C "cl program", which will look in the environment variable CL for its flags. Here is a sample ".c.obj" rule which uses this feature.

```
.c.obj:
  @ echo cl $(CFLAGS) -c $<# show what will be done
  @ %set CL=$(CFLAGS)      # set CL into the environment
  @ cl -c $<               # do the compile
  @ %set CL=               # unset CL
```

D.7 Makefile Directives

Lines in the makefile having "!" as the first character are called directives. Space is allowed between "!" and the name of the directive. There are two major classes of directives: those that affect how MAKE reads the makefile (read-time directives), and those that affect how MAKE executes shell lines (make-time directives). Some directives can be used at both read-time and make-time. For these, when the "!" is in the left-most column of the makefile line, the directive is interpreted at read-time. If the "!" is preceded by whitespace like a normal shell line, the directive is interpreted at make-time.

Here is the list of the available directives:

Directive	Type	Applicable Time
!if	conditional	read/make
!else	conditional	read/make
!elif/!elseif	conditional	read/make
!endif	conditional	read/make
!while	loop	make
!foreach	loop	make
!break	loop	make
!continue	loop	make
!end	loop	make
!include	other	read
!error	other	read/make

D.7.1 Conditional Directives

This set of directives bring classical if-then-else control to MAKE. These directives are allowed both at read-time and make-time. The general structure of this set of directives is:

```
!if condition
[!elif condition]
[!elif condition]
[!else]
!endif
```

Each !if must be matched with an !endif. There can be several !elif clauses, but at most one !else clause. Lines between !if or !elif and the next !elif, !else, or !endif make up a block. If condition is true, the block is read, otherwise it is skipped. Only the first block corresponding to the first true condition will be processed. If none of the conditions are true, and there is an !else clause, the block between !else and !endif will be processed.

D.7.2 Form of Condition

MAKE recognizes the same form of condition for !if, !elif, or !while. The general grammar is the following:

form:	expr [logical expr ...]
expr:	atom [compare atom] or ! form or (form) or { -d -e -r -w -z } atom
compare:	== != < <= > >=
logical:	&& ||
atom:	string number

In English, this means you can compare strings and numbers and combine the comparisons with logical operations. If you are a C programmer, you may recognize the grammar as a subset of C. Additionally, you are able to test the state of files.

Depending on whether an atom is a number (starts with a digit) or string (does not start with a digit), comparisons are done numerically or alphabetically. Available comparisons are: == (equal), != (not equal), < (less than), <= (less than or equal), > (greater than), >= (greater than or equal). The value of the comparison is 1 (one) if true and 0 (zero) if false.

Available logical operators are: && (and) and || (or). Comparisons "bind tighter" than logical operators so "ABC < DEF && 6 >= 5" evaluates to "(ABC < DEF) && (6 >= 5)" which is "1 && 1" which is "1" (true). Normally the form is evaluated from left to right, but parentheses can be used to modify the order. The ! (not operator) changes the sense of the comparison.

The expression "!if $(FOO)" will be true if the value of FOO is anything other than the number 0. If the value of FOO is the empty string or if it contains spaces, the expression is an error! Single quotes are used to wrap FOO up into one string. The best test for determining if FOO has a value is to use: !if ’$(FOO)’!=”.

To test whether FOO has been defined or not, the expression $d(FOO) will evaluate to 1 if it has and 0 if it has not.

The state of a file can be interrogated with the unary operators: –d (directory), –e (exists), –r (readable), –w (writable), or –z (zero length). Each operator returns 1 if the state is true and 0 if it is false. For example, !if –e filename would be true if filename existed and false if it did not.

D.7.3 Loop Directives

These directives bring looping and iteration capability to MAKE. Basically, they allow shell lines to be done a controllable number of times. These directives are only allowed in shell lines. The general structure of this set of directives is:

```
!while condition
[shell lines and directives]
!end

!foreach macro_name list_of_values
[shell lines and directives]
!end
```

In the first form if "condition" is true, the optional shell and directive lines will be executed. If "condition" is false, they will be skipped. In the second form, the macro "macro_name" will take the value of successive elements of "list_of_values" (e.g. !foreach x ${OBJS}, successively sets "x" to the members in ${OBJS}).

Each "!while" and "!foreach" must be matched with a "!end". "!break" and "!continue" cause the loop to be interrupted. "!break" causes the loop to be aborted immediately. "!continue" causes the loop to be restarted at the top. For "!while" loops, the condition is retested. For "!foreach" loops, the "macro_name" is set to the next in the "list_of_values". Here is a sample makefile which uses conditional and loop directives:

```
!if $d(CV)                         # Read-time conditional
    CFLAGS = -Od -Zi
!else
    CFLAGS = -Ox
!endif

 #  An inference rule for going from ".c" to ".obj" files (see
    section D.9).
.c.obj:
    !while 1                       # Loop forever
      -cl ${CFLAGS} -c $< 2>&1 >$*.err
      !if $(status) == 0           # Make-time conditional
        !break                     # Quit if no compiler errors
      !endif
      b -mnext_error $<            # Else start up BRIEF
    !end
    del $*.err
```

D.7.4 Include and Error Directives

!include "filename" reads in the file called "filename". If the included file is not found, a warning is issued. This directive is only in effect at read-time. "!error error_message" prints the "error_message" (everything on the line after the !error directive) and stops MAKE.

D.8 Pseudotargets

Pseudotargets are special targets which are put in MAKE description files and are used to convey additional information to MAKE. The pseudotarget name always has a period as the first character and is always in uppercase.

For pseudotargets that take arguments or have shell lines, the name of the target is required to be followed by a colon. Other pseudotargets are simply flags and are not followed by a colon.

The list of targets is:

.AFTER : — the shell lines associated with this target are run after making all applicable targets.

.BEFORE : — the shell lines associated with this target are run before making any applicable targets.

.DEFAULT : — the shell lines associated with this target are run if a target is supposed to get made but there are no rules (explicit or implicit) specifying how it gets made.

.GLOBAL_PATH — by default, the macro modifier D returns ".\" if there is no directory component to the macro being modified. With this pseudotarget defined, the current working directory will be supplied instead.

.IGNORE — shell lines returning nonzero status (i.e. the errorlevel) cause MAKE to terminate unless the pseudotarget ".IGNORE" is in the makefile or the shell line begins with "–" (hyphen). The "–i" Option gives the same effect.

.MACRO_WARN — this causes MAKE to warn about the usage of undefined macros.

.PRECIOUS : — break (control-C) and shell line errors cause the target being worked on to be deleted if it has been written to unless it's a prerequisite of the target .PRECIOUS.

.RESPONSE_LINK : — normally MAKE automatically generates response files only for Lahey OPTLINK and Library Manager. This pseudotarget allows specification of additional programs that accept OPTLINK-style response files and for which MAKE will automatically generate a response file. See section D.13 for details.

.RESPONSE_LIB : — as the previous, except for Library Manager-style response files.

.SILENT — shell lines to be executed are printed when executed unless the pseudotarget ".SILENT" is in the makefile, or the first character of the shell line is "@". Also equivalent to the –s Option.

.SUFFIXES : — this pseudotarget gives the order in which to try inference rules. See section D.9.1 for more details.

D.9 Inference Rules

MAKE can use inference rules to specify shell lines for targets for which the makefile gives no explicit shell lines. An inference rule tells MAKE how to create a target with a particular filename extension from a target with the same root name but a different extension. Inference rules look much like standard dependency lines except for having a special target name which is composed of two extensions:

```
.from_extension.to_extension :
 [WHITESPACEshell line]
```

For example, to produce an ".OBJ" file from a ".FOR" file, the inference rule could be:

```
.for.obj:
   $(FC) $<$(FFLAGS)
```

Where: Macro "FC"has the value "f77l".

Other inference rules built into MAKE are:

```
.asm.obj:
   ${AS} ${AFLAGS} $<
```

Where: Macro "AS" has the value "masm".

```
.c.obj:
   ${CC} ${CFLAGS} –c$<
```

Where: Macro "CC" has the value "cl".

```
.obj.exe:
   $(linker) $<,$@;
```

Where: Macro "linker" has the value "optlink".

D.9.1 .SUFFIXES

When MAKE searches for an inference rule, it finds the first rule where the prerequisite file exists. Since there may be several ways to produce an ".OBJ" file (e.g. from a compiler or assembler), the order in which rules are attempted is specified by the ".SUFFIXES" list. This is a list of extensions which has the default value:

```
.SUFFIXES: .exe .obj .c .for .asm
```

When looking for an inference rule, the list is searched from left to right. For example, if MAKE was trying to make "TEST.OBJ" using an inference rule, it would look first for a ".C.OBJ" rule (since ".C" is to the right of ".OBJ" in the ".SUFFIXES" list). If the rule was found, MAKE would check for file "TEST.C". If the file did not exist, MAKE would look for a ".FOR.OBJ" rule (and "TEST.FOR" file), then an ".ASM.OBJ" rule and a "TEST.ASM" file.

The example given in section D.6 could be written more succinctly as:

```
OBJS = main.obj sub.obj

test.exe : $(OBJS)
  optlink $(OBJS), $@;
```

As a result of the inference rules, "MAIN.OBJ" and "SUB.OBJ" implicitly depend upon files "MAIN.FOR" and "SUB.FOR" respectively.

The list of suffixes accumulates, so given the same ".SUFFIXES" as listed above, the effect of the following line:

```
.SUFFIXES : .exe .obj .pas
```

would be to give a combined suffix list of:

```
.exe .obj .pas .exe .obj .c .for .asm
```

A ".SUFFIXES" target with no suffixes clears the list of suffixes.

A common source of errors when using inference rules is getting the order of ".SUFFIXES" wrong. The inference rule looks like ".from.to" but the ".SUFFIXES" entry looks like ".to" [other suffixes] ".from". Note the reversal in order!

D.9.2 Inference Rule Shell Lines

When the inference rule shell line is the name of an inference rule, the shell lines corresponding to that inference rule are interpolated in place of the name.

Given the ".for.obj" and the ".obj.exe" rules known to MAKE, a ".for.exe" rule could be constructed as follows:

```
.for.exe:
    .for.obj
    .obj.exe
    erase $*.obj
```

Two optional arguments can be mentioned on the inference rule shell line. The first argument is the value $< will have during the interpolated rule, the second argument is the value $@ will have. You must supply a value for $< if you wish to supply one for $@.

When no arguments are given on the inference rule shell line, MAKE's behavior is adjust $< and $@ automatically.

NOTE: The .for.exe: rules are already known to MAKE.

D.10 Options (or Flags)

Options (also known as flags) are entered on the command line with a "–" or "/" as the first character. Options can be grouped together after a single "–", so "–dp" is equivalent to "–d –p". If a "–" appears by itself, it signals the end of the options. This can be used when the target you are making has a "–" as the first character and you do not want MAKE to misinterpret the target name as an option (e.g. "make – –readme–").

Options can be turned ON permanently by putting them in the MFLAGS macro. With this feature you do not have to continually specify commonly-used options such as –m or –D. The following DOS command will cause Options –m and –D to always be in effect:

set MFLAGS=–mD

As with other macros, the value of MFLAGS can be changed from the command line (e.g. "make MFLAGS=" means no options will be turned ON). Alternatively, the –z Option will cause MAKE to ignore the MFLAGS macro.

The –t, –i, –k, –n, –d, and –p Options are used in the sample session in section D.15. The makefile options are:

–a — all specified targets are made whether they are out of date or not.

–d — debug mode. Traces MAKE as it executes, printing the time difference between each file and the target that has it as a prerequisite.

–D — directory snapshot is enabled with this flag. The current working directory and directories along the VPATH get read into memory when MAKE starts up. Thereafter, MAKE never goes to disk (except with the "=" shell line prefix — see section D.5.6). This may cause a problem if a shell line creates a file which is not a MAKE target, and MAKE is supposed to know about the file. In this rare case, you should not use the –D Flag.

NOTE: Whenever the –n (no execute) Flag is on, the –D Flag gets turned ON automatically.

–e — environment override. Give environment macros higher precedence than makefile macros (but lower precedence than command-line macros).

–f — makefile name option. The default makefile name is "makefile" Use the –f option to specify alternate makefile names. Use the –f– option to input a makefile from the keyboard (standard input). DOS versions 3.0 and higher support using –f CON to specify the keyboard as makefile input. More than one –f Option may be used to specify multiple makefiles on the DOS command line. When using the –f– or –f CON flags for command-line input, use the **CONTROL–Z ENTER** key combination to terminate the makefile.

–h — print a help screen.

–I — ignore errors. Equivalent to the pseudotarget ".IGNORE". Causes a nonzero exit status to be ignored. Doing "make –i >errs" collects all error messages into the errs file. Interrupting "make –i" may require several **CTRL BREAKs**.

–k — keep going. When a shell line returns nonzero status, abandon work on the current target, but continue doing the make on other targets. MAKE ensures that targets which are dependent on the target in error do not get made. The result is that the maximum amount of safe work gets done.

–m — memory-miser. MAKE will reduce the amount of memory it uses to a minimum by paging itself out. With this flag, MAKE will only extract a 3000 byte overhead over executing shell lines!

–M — Microsoft make compatibility mode. This flag allows MAKE to process Microsoft Make makefiles without modification. The syntax for Microsoft make is:

    make [options] makefile

Where:

    "makefile" is the name of the Microsoft Make makefile.

This can be processed by MAKE with:

    make –M [options] makefile

–n — no execute. Display but do not execute the shell lines needed to update the targets.

–p — print information on macros, dependencies, and available inference rules. A rule is available if it is defined in a ".from.to" target and there are ".to" and ".from" extensions in ".SUFFIXES" in the correct order (".to" to the left of ".from").

–q — query. Sets MAKE's exit status to 1 if there are things to be made and 0 if there are not. If there was an error processing the makefile, the exit status is set to 2.

–r — remove inference rules. Actually clears ".SUFFIXES" after "MAKE.INI" has been read. The effect of this is to prevent MAKE from using inference rules other than those in the makefile.

–s — silent mode. Equivalent to the pseudotarget ".SILENT". Shell lines are not displayed before being executed.

–t — touch out of date targets. Rather than executing the shell lines that would update the targets, the file time of the target files is set to the current time. This will create files of zero length if they do not exist. MAKE touches files like the included "TOUCH.EXE" Program.

–z — zero. Ignore the MFLAGS macro.

–1 — one command mode. Only the very first shell line will be executed. This is useful for doing compiles from inside of editors since it spares the overhead of paging but still gives the advantage of using very little memory (less than 1000 bytes).

D.11 Multiple Directory Support: VPATH

The VPATH macro specifies the additional directory names MAKE will use to try to find files that do not have a path specification. If VPATH is not defined, only the current directory is searched for files.

One place VPATH is useful is for compiling sources for different target machines or for different memory modules. You want to keep only one set of sources, but want to compile them in a variety of ways. Only one source directory is required. Each different compilation has its own subdirectory and the makefile in these subdirectories uses VPATH to find sources in the source directory. A sample VPATH is:

```
VPATH = ..;\src\common
```

When MAKE looks for files, first the current directory is searched, then ".." (the parent directory), then \src\common. A semicolon separates VPATH directories. When MAKE tries to find a file like "TEST.C", it looks first for "TEST.C", then for "..\TEST.C" and then finally for "\SRC\COMMON\TEXT.C". The first file found is used. When MAKE is trying to find an implicit prerequisite for an inference rule, it constructs a particular file (e.g. "TEST.C"), then tries each directory in order to find the file. Needless to say, VPATH will slow MAKE down. The –D Option causes MAKE to take a snapshot of VPATHed directories and results in much faster operation.

As an example, assume "TEST.C" is in directory ".." and the makefile is:

```
VPATH = ..
test.obj:
```

When you type "make", you would get:

```
cl –c ..\test.c
```

NOTE: Although the source comes from another directory, many compilers will put the resultant ".OBJ" file in the current directory.

D.11.1 Shell Line Translation

What if you have many machine-independent files, and only a few machine-dependent files? You would like to keep the dependent ".OBJ" files in the subdirectories and all sources and independent ".OBJ" files in the parent directory. Now your makefile in each subdirectory looks something like:

```
VPATH  = ..
OBJS   = dep.obj main.obj indep.obj
DEFS   = –DIBMMONO
CFLAGS = $(DEFS) –AL
ibmmono.exe: $(OBJS); optlink $(OBJS), $@;
```

"MAIN.OBJ" and "INDEP.OBJ" are found in ".."; "DEP.OBJ" is found in the current directory; all source files are found in "..". The compilation goes smoothly, but when it comes time for the link, $(OBJS) expands to:

```
dep.obj main.obj indep.obj
```

which fails since several of the ".OBJs" are in "..". What we need $(OBJS) to expand to is:

```
dep.obj ..\main.obj ..\indep.obj
```

Therefore, MAKE translates each shell line to include the actual path to the file.

Now the link proceeds as it should with:

```
optlink dep.obj ..\main.obj ..\indep.obj, ibmmono.exe;
```

Punctuation may prevent MAKE from translating a name. If a name should be translated but it isn't, try separating the name from any punctuation with WHITESPACE. For example, when using overlays with Microsoft LINK, the object modules go in parenthesis. MAKE will not translate a name like (main.obj) but does translate (WHITESPACEmain.objWHITESPACE).

D.11.2 File Lookup

Normally, MAKE will only look on the disk for a particular file once. Thereafter MAKE keeps track of it internally. When VPATH is in effect, files can exist in directories other than the current directory and the effect of a shell line may be to move the file to a different directory on the VPATH, something MAKE does not normally consider. MAKE will only automatically look on the disk again if the target file did not exist prior to the shell line execution. Otherwise, the "= shell" line prefix (see section D.5.6) can be used to force MAKE to again look along the VPATH for the file.

As an example, assume the current ".FOR" and ".OBJ" files reside somewhere remotely (which VPATH references) and you wish to make changes to a few local copies of the sources then relink the resulting local ".OBJ" files with the remaining remote ".OBJ" files. Assume you have local copies of the required ".FOR" files but do not have local ".OBJ" files. Complete the changes and "make". The compilation will use the local ".FOR" files and will result in local ".OBJ" files. However, MAKE will not realize it should use these local ".OBJ" files to do the link because it found the remote ".OBJ" files first.

The following inference rule will force MAKE to look again along the VPATH for the target once the compile has been done:

```
.for.obj:
  = $(FC) $<$(FFLAGS)
```

D.12 I/O Redirection

In DOS, MAKE does much of its own Input/Output redirection to make it possible to redirect error messages into a file. Besides supporting the equivalent of DOS's > and >> (redirection of standard output) and < (redirection of standard input), MAKE supports the OS/2 compatible forms 2> and 2>> (redirection of standard error). For example, 2> file creates a file and sends error output to it and 2>> file opens file and appends error output to it.

Redirection of both standard output (file handle 1) and standard error (file handle 2) into a file requires forcing both file handles to refer to one file handle, which can then be redirecting into the file. The 2>&1 construct forces file handle 2 onto file handle 1. Hence, >file 2>&1 redirects standard output into file and standard error onto standard output (hence into file). This is identical to the OS/2 convention. As an example, the command:

```
f77l main.for >errs 2>&1
```

puts all normal and error output into the file called "errs". Redirection by >, >>, 2>, 2>> and < can be turned OFF by prefixing the > and < characters with backslash.

NOTE: MAKE does not support pipes (i.e. "|") and depends on DOS to do so.

D.13 Response Files

Some programs (notably Lahey OPTLINK and the Lahey Library Manager) can receive their input from a file called a response file. Usually a response file is used when the length of the command line becomes longer than DOS allows. In program development, the linker and library manager commands are used very often and MAKE generates a response file automatically when required.

D.13.1 OPTLINK Response Files

When a program name has the base name "optlink" and a command line longer than DOS allows, MAKE generates a response file and has OPTLINK use the response file. Your command:

```
OPTLINK $(OBJS), test.exe,, $(LIBS)
```

will work without modification no matter how long the line becomes.

D.13.2 Library Manager Response Files

These response files are similar to an OPTLINK response file, but slightly more complex to compensate for the Lahey Library Manager's syntax. MAKE takes the LM command:

```
lm libname [/pagesize] operator module ... [,listing] [,newlib]
```

and generates a response file with the most recent operator placed between each object module. For example:

```
lm my.lib –+ main.obj sub1.obj sub2.obj + sub3.obj;
```

becomes:

```
lm @responsefile
```

Where the response file has the contents:

```
my.lib  –+ main.obj –+ sub1.obj –+ sub2.obj + sub3.obj;
```

This is most useful when you want to update a library when some of its modules have been recompiled. For example, if "MY.LIB" depends on modules "LIBOBJS", then the following updates the library with only those modules which have been changed since the library was last written:

```
my.lib: $(LIBOBJS)
    lm my.lib –+ $?;  # $? expands to prerequisites
                      # more recent than the target
```

D.13.3 .RESPONSE_LINK and .RESPONSE_LIB

The special target ".RESPONSE_LINK" allows specification of base names in addition to OPTLINK which MAKE will recognize as accepting an OPTLINK-style response file. These programs do not have to be linkers; they just have to support a OPTLINK style response file (i.e. input lines are separated by commas; long output lines are continued onto the next output line with **+ENTER** as the last two characters on the output line).

Similarly ".RESPONSE_LIB" can be used to specify additional names for Library Manager-style response files.

For example:

```
.RESPONSE_LINK : tlink link4
.RESPONSE_LIB : plib
```

These enable OPTLINK-style response files for "TLINK" and "LINK4", and Lahey Library Manager-style response files for "PLIB".

D.14 DOS Commands, Batch Files, Redirection and Error Codes

MAKE depends on a program's exit status (a number testable with the DOS "if errorlevel" Command) to determine whether or not an error has occurred. An exit status of 0 means no error. In order for MAKE to execute internal DOS commands (like copy, chdir, etc.) run batch files, or handle pipes "|", MAKE must start a second copy of "COMMAND.COM" and have that copy execute the command.

Unfortunately, "COMMAND.COM" always reports an exit status of zero, implying no error has occurred. MAKE compensates for this by using an error file to communicate the exit status back to MAKE. The error file's name is put into the environment so your batch files can refer to it as "%MAKE_ERR%".

When MAKE needs to start up another copy of DOS's "COMMAND.COM" it builds a batch file and then has "COMMAND.COM" execute the file. If you interrupt the batch file with **CTRL BREAK**, you may get the prompt:

> ***Terminate BATCH file ? (y/n)***

Answer "y" and MAKE will know an error has occurred. After MAKE gets control back from the batch file, it detects an error by checking for the error file. If the file exists, an error has occurred; if the file is absent, the batch file finished successfully. If you use your own batch files from MAKE, they have to be modified slightly to send the exit status back to MAKE. This is

because DOS will not return to MAKE's batch file after yours has finished. The following line, placed at the end of your batchfile, is needed:

```
if exist %MAKE_ERR% do erase %MAKE_ERR%
```

To get a little fancier, your batch file could look like:

```
command1
if errorlevel 1 goto end
command2
if errorlevel 1 goto end
if exist %MAKE_ERR% do erase %MAKE_ERR%
:end
```

As you can see, the error file gets erased only if "command1" and "command2" execute without error.

D.15 Sample MAKE Session

This section describes what MAKE does when it's running. Assume you have the following makefile:

```
M         = S
CFLAGS    = –A$M
OBJS      = main.obj sub.obj
test.exe: $(OBJS)
   optlink $(OBJS), $@,, \lib\$Mlocal;
$(OBJS): incl.h
sub.obj:
   $(CC) $(CFLAGS) –Od –c sub.c
install: test.exe
   copy test.exe $(BIN)
```

and the following files are in your directory: "MAIN.C", "SUB.C", and "INCL.H". When you execute make at the DOS prompt, it first reads "MAKE.INI" then makefile. It sees the first target "TEST.EXE" and tries to make it. But first, MAKE must know if the files that "TEST.EXE" depends on are up to date. As "TEST.EXE" depends on several ".OBJ" files, and these ".OBJ" files also have prerequisites, the detailed procedure MAKE undergoes looks like this:

```
Make test.exe
  test.exe depends on main.obj and sub.obj.  Make these.
  Make main.obj
     Check for implicit prerequisites.
       Find rule prerequisites of main.obj.  Since main.obj has no
       shell lines, add the shell lines from  the inference rule.
     main.obj depends on incl.h and main.c.  Make these.
     Make incl.h
       Check for implicit prerequisites.
         Since there is no ".h" suffix in ".SUFFIXES", there are no implicit
         prerequisites.
       There are no prerequisites and incl.h exists, so it is up to date.
    Make main.c
       Check for implicit prerequisites.
         Since there are no ".from_extension.c", there are no implicit
         prerequisites.
    There are no prerequisites and main.c exists, so it is up to date.
  Compare main.obj with incl.h and main.c.  Since main.obj does not exist,
  it is out of date with respect to its prerequisites.  To update main.obj
  execute the (inference) shell line:
    cl –AS –c main.c
main.obj is up to date.
Make sub.obj
    Check for implicit prerequisites.
       Find rule .c.obj and file sub.c.
       Add sub.c to the prerequisites of sub.obj.  Since sub.obj has shell
       lines, do not add the shell lines from the inference rule.
     sub.obj depends on incl.h and sub.c.  Make these.
     Make incl.h
       MAKE already knows that incl.h is up to date.
 Make sub.c
    Check for implicit prerequisites.
       Since there are no ".from_extension.c" rules, there are no implicit
       prerequisites.
    There are no prerequisites and sub.c exists, so it is up to date.
  Compare sub.obj with incl.h and sub.c.  Since sub.obj does not exist, it is
  out of date with respect to its prerequisites.  To update sub.obj execute
  the (explicit) shell line:
    cl –AS –Od –c sub.c
sub.obj is up to date.
Compare test.exe with main.obj and sub.obj.  Since test.exe does not
exist, execute the shell line:
    optlink main.obj sub.obj, test.exe,, \lib\local;
test.exe is up to date.
```

Assuming no errors occurred, when you now execute "make" you will get the message that "TEST.EXE" is up to date. If you edit "SUB.C" and change it, when you next execute "make", MAKE will see that "SUB.C" is more recent than "SUB.OBJ" and recompile "SUB.C". MAKE will then see that "SUB.OBJ" is more recent than "TEST.EXE" and relink the files.

If you type:

make install

MAKE will ensure "TEST.EXE" is up to date, then copy it to your BIN directory. The macro BIN was assumed to be defined in your environment.

D.16 MKMF Functional Description

By default MKMF works on the file called makefile in the current directory. If this file exists, all modifications are made to it. If this file does not exist, it is created from either "MAKEFILE.PRG" or "MAKEFILE.LIB" depending on whether an executable program or library is desired (see the –l Flag in section D.19). These two files are the default template files. See section D.17 for more details.

If desired, an alternate makefile can be specified with the –f Flag and an alternate template can be given with the –t Flag.

Command-line [macros] are discussed in section D.18 and [options] are discussed in section D.10.

D.16.1 MKMF Command-line Syntax

The command-line syntax for MKMF is:

```
mkmf [options] [macros]
```

D.17 Initialization and Template Files

As with MAKE, when MKMF starts up it first reads the initialization file "MAKE.INI". This file is a good place to put the special MKMF macros which are detailed in section D.18.

When MKMF creates a makefile for the first time, it starts from a template file. The default behavior is to use one of the template files "MAKEFILE.PRG" or "MAKEFILE.LIB". As with the file, "MAKE.INI", these files are found either in the current directory or in the directory specified by the "INIT" environment variable.

D.18 MKMF Macros

D.18.1 Automatically Maintained Macros

In the process of creating and maintaining makefiles, MKMF is aware of several macros, as indicated here:

DEST — directory where the program or library is to be installed.

EXTHDRS — list of included files external to the current directory. MKMF automatically updates this macro definition in the makefile if dependency information is being generated.

HDRS — list of included files in the current directory or along the MKMF_VPATH. MKMF automatically updates this macro definition in the makefile.

LIBRARY — library name. This macro is only in library makefiles (selected with the –l Option).

OBJS — list of object files. MKMF automatically updates this macro definition in the makefile.

PROGRAM — program name. This macro is only in program makefiles.

SRCS — list of source code files. MKMF automatically updates this macro definition in the makefile.

D.18.2 Special MKMF Macros

Several macros are special to MKMF and can be used to customize this program for your compiler(s). These macros are:

MKMF_CDEFINES

Many C compilers automatically define several constants which can be used in C source files to control conditional compilation. For example, Microsoft C defines the MSDOS constant, among several others. The value of the MKMF_CDEFINES macro is a list of constants you wish MKMF to consider defined by the compiler. For ease of use with the Microsoft C compiler, MKMF predefines this macro as:

```
MKMF_CDEFINES = –DMSDOS –DM_I86 –DM_I86xM
```

Where:

> "x" is either S (small model), C (compact model), M (medium model), L (large model), H (huge model). The model type is gleaned from the –Ax Flag in the MKMF_CFLAGS macro, described next. You can define MKMF_CDEFINES to have whatever value you wish by putting it in the "MAKE.INI" Initialization File.

MKMF_CFLAGS

When using a makefile, it is conventional to put all C compiler flags in the CFLAGS macro. The flags important to MKMF are: –D (define a symbol), –I (add a directory to the include path), and –U (undefine a symbol). Besides command line flags, some compilers allow additional flags to be set in special environment variables. In particular, Microsoft C used the CL environment variable to pass flags to its "CL Program". Consequently, MKMF predefines the MKMF_CFLAGS macro as:

```
MKMF_CFLAGS = $(CL) $(CFLAGS)
```

When generating dependency information, MKMF considers local directories to be the current directory plus directories specified in the MKMF_CFLAGS macro with the –I (or /I) C compiler option. Thus, you may wish to add additional local directories to MKMF_CFLAGS:

```
MKMF_CFLAGS = $(CL) $(CFLAGS) –Ic:\common
```

MKMF_FLAGS

The MKMF_FLAGS macro can be used to modify the operation of MKMF. The value of this macro is a list of MKMF options (see the **MKMF Options** section D.19) such as might be entered on the command line. For example, the following would specify verbose output and local dependency information:

```
MKMF_FLAGS = –v –d L
```

NOTE: MKMF will ignore MKMF_FLAGS if the –z Option is given on the command line.

MKMF_CGLOBAL

This macro specifies which include directories are to be considered global. The macro has the same form as the Microsoft C INCLUDE environment variable (i.e. a list of directories separated by semicolons). That is:

```
MKMF_CGLOBAL = directory–1;directory–2[ ; ... ]
```

For ease of use with Microsoft C, this macro is predefined as:

```
MKMF_CGLOBAL = $(INCLUDE)
```

MKMF_SUFFIX

This macro tells MKMF how to treat files of a particular filename extension. The macro is covered in detail in section D.18.2.

MKMF_VPATH

The value of this macro is the name of directories additional to the current directory in which to look for source files. MKMF predefines this macro as:

MKMF_VPATH = $(VPATH)

D.18.3 Macro Replacement

MKMF will also replace the value of any macro in the makefile if a new macro value is given on the command line. Thus if CFLAGS is "–Os" in the makefile, 'mkmf CFLAGS="Od –Zi"' will change its value in the makefile to "–Od –Zi". Note the use of double quotes around command line macros that include spaces. MKMF will only change the value of macros already existing in the makefile. It will not add new macros.

D.19 MKMF Options

The MKMF Options are case sensitive. The meaning of the options that modify MKMF operations are:

–c — C-mode. When C sources or C include files are processed, the C preprocessor directives (i.e. #include, #define, #undef, #ifdef, #if, #elif, and #endif) are interpreted and acted upon accordingly. The MKMF_CDEFINES (see section D.18.2) macro is used to give MKMF the set of symbols predefined by the compiler.

–d G — global dependencies. Generate global dependency information. When generating dependencies, use the current directory plus those mentioned in the MKMF_CFLAGS and MKMF_CGLOBAL macros. See section D.18.2 for more details.

–d L — local dependencies. Generate local dependency information. When generating dependencies, use the current directory plus those mentioned in the MKMF_CFLAGS macro.

–d N — no dependencies. No dependency information is determined. This is the default.

–f — makefile name option. The default makefile name is "makefile" Use the –f option to specify an alternate makefile name.

–h — help. Display a help screen.

–I — interactive. Ask for the name of the PROGRAM or LIBRARY to be used, and the directory it is to be installed into.

–l (small L) — library. Create a library makefile (using "MAKEFILE.LIB" as the template). The default is to create a program makefile (using "MAKEFILE.PRG" as the template). This flag only has affect when a makefile is being created from a default template.

–s — slow. Normally, MKMF will read each file at most once. For example, if both "MAIN.C" and "SUB.C" include file "STDIO.H", then when processing "MAIN.C", "STDIO.H" will also be read. However when processing "SUB.C", "STDIO.H" will not be read again. –s tells MKMF to skip this optimization and to read a file each time it is encountered. This option is only useful in conjunction with the –c (C-mode) Option.

–t — template name option. This option uses one argument to specify the name of an alternate MKMF template filename instead of using the default templates, "MAKEFILE.LIB" and "MAKEFILE.PRG".

–v — verbose. Display the actions of MKMF as it generates dependencies.

–z — zero. Zero MKMF_FLAGS. This option causes the MKMF_FLAGS macro to be ignored.

D.20 Dependency Generation

If Option –d L or Option –d G is given, MKMF automatically generates dependency information for the sources it finds in the current directory and directories named with the MKMF_VPATH variable. It determines which files are sources based on the file name suffix (i.e. extension). Currently, MKMF knows about the following suffixes:

Suffix	File Type
.c/.c++	C/C++ source
.for/.f	FORTRAN source (see note in D.22)
.h/.h++	C/C++ include files
.obj	object files
.pas	Pascal source
.asm	assembler (MASM) source

Generated dependency information appears after a line in the makefile beginning with "### OPUS mkmf: ...". Any information below this line will be replaced whenever dependencies are recalculated by MKMF. If the line does not exist, it is added to the end of the makefile.

D.21 Include Files

When looking through source files to find dependency information, MKMF looks for include files. These are files that the compiler includes in your sources. Include files can be found in a directory other than the current directory and the way MKMF finds them depends on the type of include. Here is a list of the include types supported by MKMF, their syntax, and the method used to find the include files.

D.21.1 FORTRAN Include Types

Syntax:

```
INCLUDE filename
INCLUDE 'filename'
INCLUDE "filename"
```

FORTRAN include files are looked for in the current directory.

NOTE: Include files must not use the ".FOR" or ".F" extensions in order for MAKE file inference rules to work.

D.21.2 C Include Types

Syntax:

```
#[WHITESPACE]include "filename"
#[WHITESPACE]include <filename>
#[WHITESPACE]include NAME
```

Where:

"#" must be the first character on the line, although in accordance with the new ANSI C standard, it does not have to be in the leftmost column. The third form is compatible with ANSI C when the NAME is a preprocessor symbol and its value is consistent with one of the first two forms.

For C, include files between angle brackets (i.e. <filename>) are searched for in the standard include directories. In MKMF, the MKMF_CGLOBAL macro names these directories.

C compilers also accept flags of the form –Idir or /Idir to add directory "dir" to the standard include directories and MKMF looks in the MKMF_CFLAGS macro for flags of this type. For include files between double quotes (i.e. "filename"), first the current directory is searched, followed by the standard include directories.

When calculating dependencies by reading include files, –d G tells MKMF to search the current directory, directories specified with the –I (or /I) Flags found in the MKMF_CFLAGS macro, and directories specified in the MKMF_CGLOBAL macro. If –d L is given, directories specified in the MKMF_CGLOBAL macro are not considered. If –d N is specified, no scanning of source files occurs.

NOTE: C++ has essentially the same syntax as C and the same include type.

D.21.3 PASCAL Include Types

Syntax:

```
#[WHITESPACE]include "filename"
#[WHITESPACE]INCLUDE "filename"
```

Where:

"#" must be the first character in the line.

PASCAL include files are looked for in the current directory followed by directories mentioned in flags of the form –Idir in the PFLAGS macro.

D.21.4 ASM Include Types

Syntax:

```
include filename
INCLUDE filename
```

ASM include files are looked for in the directories mentioned in flags of the form –Idir in the AFLAGS macro followed by the current directory.

D.22 The MKMF_SUFFIX Macro

MKMF can be instructed to recognize additional filename suffixes, or ignore ones that it already knows about, by specifying suffix descriptions in the MKMF_SUFFIX macro definition. Each suffix description takes the form:

.suffix:tl

Where:

"t" is a character indicating the contents of the file

"s" = source file;
"o" = object file;
"h" = header file; and
"x" = executable file; and

"l" is an optional character indicating the include syntax for included files:

"C" = C syntax;
"F" = FORTRAN syntax;
"A" = ASM syntax; and
"P" = PASCAL syntax.

The following table describes the default configuration for MKMF:

Suffix Description	**File Type, Include Type**
.asm:sA	source, assembler
.c:sC	source, C
.cxx:sC	source, C
.for:sF	source, FORTRAN
.f:sF	source, FORTRAN
.h:hC	include, C
.hxx:hC	include, C
.obj:o	object — currently there can be only one suffix that corresponds to object files.
.pas:sP	source, PASCAL

For example, to change the object file suffix from ".OBJ" to ".O", prevent FORTRAN files from being scanned for included files, and add a suffix ".INC" for ASM header files, the MKMF_SUFFIX macro definition would look like:

```
MKMF_SUFFIX = .o:o .for:s .inc:hA
```

A convenient place to define MKMF_SUFFIX is in "MAKE.INI".

D.23 The TOUCH Utility

The included program "TOUCH.EXE" is used to modify the time of a file. The way to invoke it is:

touch [–cfv] [–d mm-dd-yyyy] [–t hh:mm:ss] [–i infile] file ...

By default, touch sets the time of the specified file(s) to the current system time. If the file exists, the touch will occur if there is write permission or if –f (force) is chosen. If a file does not exist, it will be created with zero length unless –c (no create) is specified. The –v Flag turns on verbose mode.

–d and –t allow specification of the date and time to be used instead of the system date and time. The format of the date and time options is like that of the DOS Date and Time Commands. For the date, yyyy may be 2 digits in which case 19yy is inferred. Partial dates and times can be used (e.g. –t 8 sets the time to 8:00:00).

The –i Flag indicates that the date and time information is to come from the date and time of infile. This flag provides a convenient method of copying timestamps to related files.

Wildcards are accepted in the file names.

D.24 The RSE Utility

Normally, DOS cannot redirect standard error messages into a file. The included utility "RSE.EXE" (Redirection of Standard Error) causes messages destined for standard error to instead be sent to standard output, which can be redirected. The usage is:

rse program [arguments] [> file]

That is, do whatever you would normally do to run a program, except preface the entire line with RSE.

D.25 MAKE Error and Warning Messages

When MAKE encounters a problem, it produces either an error message or a warning, depending on the severity of the problem. Warnings cause a diagnostic message to be printed out, followed by the text string "(warning)".

Errors also print out a message, followed by the string "Stop.", but MAKE also aborts. Note that before aborting, any shell lines associated with the ".AFTER" target get executed.

The style of the diagnostic message changes according to when the error occurs. During make-time the style is simply:

> ***MAKE: message.***

Where:

> "message" is the diagnostic message.

At read-time, the style is:

> ***MAKE: file (line number): message.***

Where:

> "number" indicates the line in file "file" that produced the diagnostic. The additional information available during read-time can be helpful when debugging your makefiles.

When a message involves a makefile directive, the message starts with the directive character. The following is an alphabetical listing of the warning and error messages that MAKE may produce. Along with each message is a description of the problem along with possible corrective actions.

D.25.1 MAKE Error Messages

bad foreach 'line'

The "foreach" line has an incorrect syntax. The correct syntax is:

    !foreach macroname [in] listofvalues

break/continue outside of while

A break or continue has occurred without the introductory while or foreach.

can't open batch file 'file'

When executing multiple commands (section D.5.7) on a single shell line, the commands are written to a batch file and the batch file is then executed. The probable cause of this is that the MAKE_TMP variable has not been set properly.

can't open filename for input/output

The redirection of input and/or output was specified incorrectly.

don't know how to make 'target'

This is one of the most difficult to understand error messages. In essence it means MAKE has exhausted all possible means of making target. This means that target does not exist and it has neither explicit prerequisites (named on a dependency line), nor implicit prerequisites (computed on the basis of inference rules). It is harder to determine why there are no implicit prerequisites and the "–d" (debugging information) Flag (section D.10) can be very helpful in tracking down the cause.

dup2(stdin/stdout/stderr) failed

The redirection of input and/or output could not occur. This is an internal error and should be reported to Lahey.

else/endif without matching if

An else or endif has occurred without the introductory if.

end without matching while/foreach

An end has occurred without seeing an introductory while or foreach.

!error: message

Your makefile used the error directive to terminate MAKE.

if/elif/else from line number is unterminated

The if, elif, or else indicated in line number was not balanced with an endif before the end of the makefile was read.

if nesting too deep. Max 20 levels

Only 20 levels of if nesting is supported, and your makefile has more than this.

MAKETMP (=value): no \ for DOS < 3.0

This message means the value of the MAKETMP variable is not allowed to contain any path separators in it for DOS older than 3.0.

maketmp failed

This is an internal error and should be reported to Lahey.

missing character in "name"

MAKE was looking for character to demark the end of macro name but got to the end of the line instead.

MSPAWN: Fatal error: can't realloc memory

This serious error can occur only when using the Memory-Miser (section D.10). MAKE stops instantly because it can't recover the memory it freed to DOS. The probable cause is damage to DOS's memory area caused by the program that MAKE was trying to execute. Please report this error to Lahey.

MSPAWN: Fatal error: can't read paging file

This error can occur only when using the Memory-Miser (see section D.10). MAKE stops instantly because it can't recover the memory it had written to the paging file. A possible cause is that the paging file was accidently deleted.

need a filename after –f

The –f Flag requires the name of a file as the next thing on the command line.

nothing to make

MAKE has nothing to do because no target was specified on the command line, and no target exists in the makefiles.

only 20 makefiles allowed

Only 20 makefiles can be specified on the command line with the –f Flag.

out of memory

MAKE ran out of memory in its dynamic memory pool. You may have to switch to "MAKEL.EXE", the large data model version of MAKE. If you are already using this version (you can check by saying "make –h"), you may have to break up your makefile into smaller makefiles.

recursive macro "name"

The macro called name has a recursive value. That is, the value of the macro refers to itself. For example, trying to determine the value of MACRO where MACRO = $(MACRO) is recursive. You can use the –p Flag (print make-file information) to show the value of the macro and where the macro is defined.

shell line without target

There is no target to which this shell line belongs. Shell lines are indented from the left column by a tab or space character.

test: message "where"

There was an error in the conditional (if) tester. The "where" is the position in the line at which the error occurred. Please refer to section D.14 for details on the acceptable syntax. Here is the list of messages and their meanings:

bad expression

The expression was not parsable.

string expected; got text

As indicated, a string was expected but not seen.

trailing junk

The expression is finished, but there was more text on the line.

unknown operator

The operator was not recognized.

")" expected; got text

A right parenthesis was expected but not seen.

TOK stack under/overrun

An internal error which should be reported to Lahey.

while nesting too deep. Max 10 levels

Only 10 levels of while nesting is supported, and your makefile has more than this. Reduce the level of while nesting.

D.25.2 MAKE Warning Messages

"target" unmade because of prerequisite errors

The –k Flag was being used to do the maximum useful work. One of the prerequisites for the current target was not made because of an error, so the current target cannot be made with confidence.

can't exec "shell_line"

The program named on the shell_line could not be executed. Following this warning is an error message which indicates why the program could not be executed. Sometimes the program name is spelled wrong or the program could not be found on the directories named in the PATH environment variable. Another common error is that there is not enough memory to execute the program. In this case, try using the Memory-Miser feature exclusive to MAKE (section D.10).

can't free paging memory

It was impossible to free memory correctly in order to use the Memory-Miser. Your only recourse is to stop using the Memory-Miser for this makefile. Please report this warning to Lahey.

can't have : and :: dependencies for "target"

The target was given on the left side of a regular (single colon) dependency and also on the left side of a double colon dependency. These can not be mixed. See section D.5.4 for more details on double colon dependencies.

can't open response file "name"

An automatically created response file could not be opened for writing. Please report this warning to Lahey.

can't read included file "name"

The file indicated with the !include directive does not exist.

can't read makefile "name"

The makefile indicated with the –f Option does not exist.

can't touch "file"

MAKE could not "touch" (i.e. modify the filetime) of file. If the file is a regular file, the next line indicates why the "touch" did not happen.

can't write paging file file

It was not possible to write memory to the paging file. Check to make sure that file is a legal filename and that there is sufficient storage on your disk for the file (about 70K in size).

can't write response file "file"

The response file couldn't be opened for writing.

macro "name" has no value

The make-time macros @, <, ? and ** initially have no value and if you use them before they have had their value set properly, you will see this message.

macro "name" not found

The macro called "name" is undefined. This message is only seen when the .MACRO_WARN (section D.8) pseudotarget has been used. Otherwise, MAKE quietly uses the null string for the macro's value. When trying to debug your makefiles, this pseudotarget can help catch simple errors in spelling.

redirect error: file

Redirection failed because file couldn't be opened in the required mode. This message is followed by the reason the file couldn't be opened. Check the writability of the file if you are trying to redirect into it.

removing target

The target is removed because it had been modified when an error occurred in one of its shell lines. This is to prevent partially written files (such as object files) from being left behind when a compilation is aborted.

too many shell lines for "target"

A target mentioned on a regular (single colon) dependency line can only be given shell lines one time. The target can be mentioned in more than one dependency line, but any additional appearances only indicate that the target has additional prerequisites. Additional shell lines will be ignored.

unknown option "option"

Only the options listed when you do a "make –h" are acceptable to MAKE.

unknown pseudotarget "name"

Only those pseudotargets listed in section D.8 are valid.

D.25.3 MKMF Warning Messages

OPUS MKMF: filename (line number): can't find include_file

When MKMF can't find included file include_file it reports the position (line number) and file (filename) of the first place the include file is encountered.

OPUS MKMF: Can't generate OBJS list without an object suffix

There must be exactly 1 suffix with the "o" (object) file type. If there isn't one, MKMF won't be able to update the OBJS macro nor will it search for dependencies.

D.26 MAKE Glossary

This glossary contains short definitions of some of the more technical terms used in MAKE.

archive This is a collection of modules, usually object modules. We use the term to describe Lahey Library Manager and LIB (Microsoft Library) files.

cl The name of the Microsoft C compile-and-link program. Other C compiler manufactures have similar programs: tcc (Borland) and wcl (WATCOM).

dependency line The relationship between targets and prerequisites is indicated in a dependency line. As an example:

```
target : prerequisite-1 ... prerequisite-N
```

This means that target depends on each of prerequisite-1 through prerequisite-N. When making target, MAKE first ensures each of the prerequisites is made. If any of the prerequisites is newer than target, then target gets updated.

directives These are used to modify the way MAKE reads the makefile and how behaves at make-time. All directives have the directive character (!) as the first character on the line. A sample directive is !include, which causes MAKE to read in a named file.

DOS The predominant operating system on IBM PCs, ATs and compatibles, produced by Microsoft Corporation.

explicit prerequisite A prerequisite mentioned explicitly in a dependency line. For example, "DEF.H" is an explicit prerequisite of "MAIN.OBJ" in the following:

```
main.obj: def.h
```

implicit prerequisite This is a prerequisite which is determined with the use of an inference rule. For example, if "MAIN.OBJ" was being built and the ".C.OBJ" inference rule was being used to do so, then "MAIN.C" is an implicit prerequisite of "MAIN.OBJ".

inference rule MAKE can infer the prerequisite for a particular module with an inference rule. This rule lists the shell lines for making a module of a particular extension (or suffix) from a module of another extension. For example, the standard rule for making an ".OBJ" file from a ".FOR" file is:

```
.for.obj:
 $(FC) $<$(FFLAGS)
```

The order inference rules are attempted is given by the order of the suffixes in the ".SUFFIXES" pseudotarget.

INIT This environment variable names the directory that MAKE and MKMF use as the default location for "MAKE.INI", MAKEFILE.PRG", and "MAKEFILE.LIB".

LM The name of the Lahey Library Manager, which maintains module archives. These archives can also be produced by Microsoft Librarian, TLIB (Borland) and OPTLIB (SLR Systems). MAKE has special handling for LIB-compatible programs to overcome limitations in DOS.TLIB

LINK The name of the Microsoft Linker, which produces executable programs from object files and object archives. Other programs which work like LINK include TLINK (Borland) and Lahey OPTLINK (SLR Systems). MAKE has special handling for Microsoft LINK-compatible programs to overcome limitations in DOS.

module This refers to a single file, such as a source or object file, which is combined with other modules to build a project, such as an executable.

null string The null string is a string with no characters.

OS/2 Another operating system produced by Microsoft Corporation.

PATH This environment variable is used by DOS to indicate the order of directory searching for executable files. For example, with PATH=c:\bin;c:\utils, DOS will look for an executable program first in the current directory, then in directory c:\bin, then in directory c:\utils.

prerequisite A target which must be updated prior to making the current target. Prerequisites show up to the right of the colon on dependency lines.

pseudo-targets These are targets which have names of the form ".NAME" and have special meaning to MAKE. For example, ".SUFFIXES" lists the order of suffixes to try when using inference rules.

quoting character A quoting character is a character (for MAKE, the backslash, \) used to remove the significance of other special characters. For example, the character that introduces a comment in a makefile is the pound sign, #. Usually, everything on the same line after this character is ignored. However, if you must have a # somewhere in your makefile, you can use the quoting character to remove the special character of #. For example:

```
FOO = 1 2 3 # 4 5 6
```

results in "FOO" having a value of "1 2 3", but

```
FOO = 1 2 3 \# 4 5 6
```

results in "FOO" having a value of "1 2 3 # 4 5 6".

shell lines These are the commands that will be executed by the operating system to update a target which must be made. Under DOS, internal commands such as DIR, COPY, CHDIR, etc., are run by executing the shell indicated by the COMSPEC environment variable. Other commands are executed directly by starting up the indicated ".EXE" or ".COM" file.

standard error/output There are two output streams (or file descriptors) which write to the console. Standard output is usually used to output general messages. Standard error is usually used to output error messages. Under DOS it is only possible to redirect standard output into a file. The utility "RSE.EXE" can be used to redirect standard error messages into standard output messages, which can then be redirected by DOS.

target A target is a "thing" that we can make. Often targets are in one-to-one correspondence with files. Before a target gets made, all of its prerequisite targets must be made first. Because of this, a single target can be used to name a list of other targets to be made. For example,

```
dummy: target1.exe target2.exe target3.exe
```

timestamp This is the time and date information stored with a file in the directory.

RESERVED FOR YOUR NOTES

E LAHEY LIBRARY MANAGER

This appendix describes the operation of the Lahey Library Manager, which allows you to create, modify and maintain libraries by adding object files, and by copying, moving, deleting and/or replacing object modules.

Libraries often contain frequently-used routines and are the result of combining separately-compiled object files into a single library file. The Lahey Library Manager can also read an object file, separate the routines into modules and then generate a new library containing these modules.

Libraries eliminate the necessity of recoding functions each time they are needed. A program that has been linked with the required library files can use the routines from the library exactly as if they were included in the program.

The default extension for libraries is ".LIB", although any extension not specifically reserved for other programs is acceptable. Object files included in a library are referred to as "Object Modules". Object files and object modules must be considered distinct entities. An object file can exist independently of a library and can be given an extension and a path on any valid drive. The default extension for object files is ".OBJ". An object module only exists as a part of a library and can have neither an extension nor a pathname.

The Lahey Library Manager creates an index of all routines and public symbols used in a library so the linker can have fast access to the required routines without having to search the entire library sequentially.

E.1 Library Manager Input

This section provides an overview of the library manager commands and options. It also provides detailed descriptions of the LM library management commands and options used for any of the three input methods listed below.

E.1.1 The LM Command

The Lahey Library Manager is invoked with the LM Command on the DOS Command Line in one of three methods:

1. Prompt Response Input;

2. Command Line Input; and

3. Response File Input

E.1.2 Option Switches Overview

The Lahey Library Manager provides three option switches. The slash (/) that precedes each option switch is the option delimiter and must be used. The option switches are:

- **/PAgesize**

- **/EXtractall**

- **/Help**

The portions of these switches shown in upper case letters are the valid abbreviations for the switches. The position of the option input field in the LM Command Line syntax is shown in bold below:

LM <old library name>[**/options**][commands][,[list filename][,[new library name]];

E.1.3 The /PAGESIZE Option Switch

The /PAGESIZE Option Switch allows you to specify the page size (in bytes) of a library. A library's page size affects the alignment of library object modules by defining the starting point for these object modules in multiples of the page size. The default page size is either 16 bytes or the page size of the existing library. The valid page size range is 16 bytes to 32,768 bytes, inclusive, in integral powers of 2 only.

The syntax for the /PAGESIZE Option Switch is:

```
LM <old library name>/PAgesize:number of bytes;
```

/PAGESIZE Example:

```
LM libname/PA:32,,newlib;
```

This example creates the **newlib** library with the contents of the **libname** library but with a pagesize of 32 bytes.

The library page size can be defined by using the /PAGESIZE Option when you create or modify a library. A non-existent library specified in the old library name field followed by the /PAGESIZE Option, the Add Modules Command and a semicolon, results in a new library being created with the specified modules and the desired page size.

NOTE: Be sure to use a colon between the "/PA" and the number of bytes. Also, do not enter any spaces between the "/PA" Option and the byte number.

The larger the page size defined for a library's object modules, the larger the library can be. The upper bounds of library size is determined by the page size number * (times) 65,536. For example, a library with a page size of 16 bytes must be 1MB or smaller (16 * 65,536 bytes). A smaller page size results in smaller library file length and in the least waste of disk memory.

E.1.4 The /EXTRACTALL Option Switch

The /EXTRACTALL Option Switch extracts all object modules from a specified library. Once extracted, these object modules (which become object files) can be added to another existing library with the Add Modules Command. This process has the same effect as merging the contents of two distinct libraries. The extracted object files can also be linked with your program if you want to use the routines directly and not call a library. The syntax for the /EXTRACTALL Option Switch is:

LM <old library name>/EXTRACTALL;

Use of this option requires no other input. The /EXTRACTALL Option Switch performs like the Copy Modules Command in that it does not delete the extracted modules from the library when it writes them to object files.

E.1.5 The /HELP Option Switch

The /Help Option Switch prints a summary of the proper library manager command syntax and option switch syntax to the console, then exits. The syntax for the /HELP Option Switch is:

LM/HELP

or

LM/H

E.1.6 Library Manager Commands

The Library Manager Commands are entered in the command field following the old library name and options fields on the LM Command Line. Each command symbol is an input delimiter, so no commas or spaces are required between commands. Refer to Section E.5 for detailed descriptions and examples of each command's syntax and usage. The position of the command input field in the LM Command Line syntax is shown in bold below:

LM <old library name>[/options]**[commands]**[,[list filename][,[new library name]];

The LM Commands are:

+ **The Add Modules Command:** The addition symbol appends the object file(s) following each + symbol to the end of a library. Pathnames can be used if the object file is not in the current directory. The library manager assumes the extension ".OBJ", if none is specified. Use the Add Modules Command to merge the contents of two libraries by adding an existing library to the one specified in the old library name field. The existing library to be added to the library specified in the old library name field must have the ".LIB" extension as part of the input.

– **The Delete Modules Command:** A subtraction symbol preceding an object module name deletes that module from the library specified in the old library name field. Object module names do not have extensions and cannot be given pathnames.

NOTE: DOS allows the "–" symbol in filenames but LM will parse this symbol as the Delete Modules Command. Therefore, we recommend that you not use the "–" symbol in filenames. If a filename has this symbol, rename the file using an underscore or other character instead of a "–".

–+ **The Replace Modules Command:** The subtraction and addition symbols combine to form the Replace Modules Command. The Replace Modules Command, followed by an object module name, replaces that module by deleting it from the library and then appending an object file with the same name as the deleted module to the library. The file specified to replace the deleted module must exist before performing the replacement. The module to be replaced can have no extension and no pathname. The object file that replaces the deleted module is assumed to have the ".OBJ" extension and to reside in the current directory unless specified otherwise. This command can be used to update existing library object modules that have been modified rather than having to create a new library just to include a modified module.

* **The Copy Modules Command:** The asterisk is the Copy Modules Command symbol. An asterisk in front of an object module name copies that module from a specified library file into an ".OBJ" file with the same name. The original library remains unchanged. The library manager automatically adds the assumed extension ".OBJ", the drive letter and the current directory's pathname to the copied object module's name to form a valid object filename. The ".OBJ" extension, drive letter and pathname for the object file are not optional and cannot be changed. If you need to change the object filename, extension, drive letter or pathname, use the DOS Copy or Rename Commands.

–* **The Move Modules Command:** The subtraction symbol followed by an asterisk forms the Move Modules Command which moves a specified object module from a library to an object file of the same name. This operation is like the copy operation above, except that it deletes the specified object module from the library.

E.1.7 Terminating Library Manager Operations

Entering **Control C** (pressing the **control** and **c** keys at the same time) interrupts library manager processing and returns the DOS Command Line Prompt.

The next sections describe the three library manager input methods and the use of these commands and options for each type of input.

E.2 Prompt Response Input

This method of using the library manager involves responding to a series of library manager-generated prompts. This interactive input method is invoked at the DOS Command Line Prompt by entering:

LM

and pressing the **RETURN** or **ENTER** Key. The library manager responds by displaying a sequence of prompts in the following order:

Library name:
Operations desired:
List filename:
Output library name:

These prompts are actually displayed one at a time. The library manager requires the responses to be in the proper format for each prompt and in the order listed above. Pressing a comma or the **RETURN** or **ENTER** Keys at a prompt forces LM to use the default for that input (if there is one) and the next prompt is displayed. Below is a summary of the input required for each prompt:

Library name: This prompt requires the name of the library to perform the manager functions upon. This input is identical to the old library name input field for the Command Line Input Method described in Section E.3.1. If either the /PAGESIZE, /EXTRACTALL and/or /HELP Option Switches are needed, they must be entered in response to this prompt immediately after specifying the old library name. There is no default for this prompt. You must always specify a library name in this field.

Operations desired: This prompt requires the input of any of the library manager commands explained in section E.2.6. Press the **ENTER** or **RETURN** Key if no LM Commands are required. There are no defaults and there is no required input for this prompt. If no operations are specified in response to the **Operations desired:** prompt, LM performs a module consistency check and no new library is created.

List filename: This prompt requires input for the list filename if one is desired. Press the **ENTER** or **RETURN** Key if no list file is required. The default is for the library manager to use a list filename of "NUL" which suppresses creation of a list file.

Output library name: This prompt requires the same input as for the new library name field described in Section E.3.3. Press the **ENTER** or **RETURN** Key if no new library name is required. The default is to use the old library filename for the new library name if none is specified in response to this prompt. LM copies the source library to one with the same name but with the ".BAK" extension. This only occurs if at least one operation is performed on the source library.

E.2.1 The Ampersand Input Line Extender

If you require many library manager operations and/or need to specify more object modules or object files than will fit on one screen line when using either prompt response or response file input, use the ampersand symbol (&) to continue the input on the next screen line. The ampersand causes the library manager to read the line following this symbol as an extension of the previous line of input. The ampersand symbol is valid when used after an object module name or an object filename.

Do not use the ampersand symbol between a library manager command symbol and an object module or object filename when continuing input for more than one line on the screen.

LM Command Line Ampersand Extender Example:

```
LM  oldlib +module1&
+module2 &
+module3 &
+module4,;
```

This results in **modules 1,2,3** and **4** being added to the end of the **oldlib** library. No listing file or new library is created.

E.3 Command Line Input

Another way to invoke the Lahey Library Manager is the command line input method. The command line syntax is shown in the example below. This section presents information on each element of the command line in the order it appears.

To use the command line input method, enter the command "LM" followed by input in the order listed in the example below. The input examples are enclosed in brackets to show that they are optional. The part of the example in the "<>" brackets is required, except that you must replace the term "old library name" with the name of the library you intend to modify.

LM <old library name>[/options][commands][,[list filename][,new library name]];

Commas must separate each input field except between the old library name field (and options) and the first LM command (if any). If you intend to omit a type of input, you must still use the comma to separate any blank fields. The library manager has default values for each of these input fields except the old library name field and the commands field. The defaults are described in the prompt response input method – Section E.2. These defaults are specified by entering a semicolon (;) after the last input field you explicitly enter.

For example, to use all defaults, the syntax would be:

LM <old library name>[/options];

The input fields and their meanings are as follows:

1. **old library name** – the name of the library to create or modify. If you use an option, the option follows the library name before the semicolon.

2. **/options** – There are three LM options: /PAGESIZE to specify the library page size, /EXTRACTALL to extract all of a library's object modules to make it possible to merge the contents of two libraries and/or /HELP to see LM command and option syntax. Sections E.1.3 to E.1.5 explain these option switches.

3. **commands** – any of the following library commands:

   + the Add Modules Command;

   – the Delete Modules Command;

   –+ the Replace Modules Command;

   ***** the Copy Modules Command; and

   –* the Move Modules Command.

These command symbols, their functions and usage syntax are explained in Sections E.1.6 and E.5.

4. **list filename** – used to specify the cross-reference-listing filename.

5. **new library name** – used to specify the new library name.

NOTE: A semicolon as the last character on any command line designates the end of user input. If the semicolon is entered before all elements of the command line are specified or forced into the default mode with commas, then LM performs the specified operations and then displays prompts for the remaining required input. If no commands, list filenames or new library filenames are entered, LM performs a consistency check on the modules in the library specified in the old library name field.

E.3.1 The Old Library Name Input Field

The position of the old library name input field in the LM Command Line syntax is shown in bold below:

LM **<old library name>**[/options][commands][,[list filename][,[new library name]];

The old library name input field is used to specify one of three things:

1. The name of the existing library to modify; or
2. The name of the library to serve as a source for creating a new library with a different name; or
3. The name of a new library to create.

The default filename extension for libraries is ".LIB". The extension may be omitted if it is ".LIB". If your old library filename extension is something other than ".LIB", be sure to specify the library's extension or the library will not be found. In this case, LM accepts the new library name and then displays the series of four prompts with the cursor following the **Operations desired:** prompt.

Enter the required prompt response input if you want to create a new library or press the **control** and **c** keys at the same time to abort the process.

E.3.2 The List Filename Input Field

The list filename field is used to specify a filename for the output of a cross-reference-listing. The position of the list filename input field in the LM Command Line syntax is shown in bold below:

> LM <old library name>[/options][commands][,**[list filename]**[,[new library name]];

A cross-reference-listing includes:

- An alphabetical list of the public symbols defined in a library. The public symbols list shows in which library object module the symbols are referenced.

- A list of all object modules contained in the library. Following each object module name is an alphabetized list of public symbols defined in that object module.

There are no reserved or default list filename extensions. If an extension is not specified, the output file will not have one. If a list filename is not specified, the cross-reference-listing is not generated. The list filename field can include a pathname if the output is to be sent to a destination outside of the current directory. Specifying "CON" in the list filename field sends the file output to the screen. Specifying "LPT1", "LPT2" or "PRN" in the list filename field sends the file output to a printer.

E.3.3 The New Library Name Input Field

The position of the new library name input field in the LM Command Line syntax is shown in bold below:

> LM <old library name>[/options][commands][,[list filename][,**[new library name]**];

The new library name input field is used to specify the library to create as a result of the specified library manager operations. The default library filename extension is ".LIB".

The next section describes the LM response file input method.

E.4 Response File Input

Response files perform a predefined set of library manager operations, eliminating the need to constantly repeat the same LM Command Line input or prompt response input.

Use an ASCII text editor to create a response file containing the instructions required to perform the desired library manager operations. The input in the response file must be in the same sequence and have the same syntax as the required input in a command line or in response to the prompts shown in the previous sections. The ampersand continuation line symbol can be used to extend the operations requested in the commands input field to more than one line in the response file.

To invoke the response file type:

LM @<response filename>

The response filename is the name given to the file created with a text editor containing the sequence of instructions. The response filename can have a drive letter, pathname and extension different than the currently active directory. The response filename must only be entered on the library manager command line. The response file input is handled in the same manner as command line input. A carriage return or line feed after each item in the response file is considered identical to pressing the **ENTER** Key in response to a prompt or using a comma delimiter on the command line. Do not omit any lines of input in the response file. The response file input is echoed to the screen showing the prompt responses being processed by the library manager. Use a carriage return to force any defaults or to omit any input in the response file. LM's defaults are defined in Section E.2.

Response File Example:

Oldlib<CR>
+object1+object2–module3*module4<CR>
Listfile.pub<CR>
Newlib<CR>

This response file causes the library manager to perform the following operations, in the following order:

1. The object module **module3** is deleted from the copy of the **Oldlib.lib** library used to create **Newlib.lib**;

2. The object files **object1** and **object2** are added to the next two positions in the **Newlib.lib** library;

3. The object module **module4** is copied into an object file named **module4.obj;**

4. The cross-reference-listing file **Listfile.pub** is created; and

5. A new library called **Newlib** is created containing **object1**, **object2** and **module4**.

The <CR>s in the example above represent carriage returns.

E.5 Library Manager Functions

This section provides detailed examples on how to use the Lahey Library Manager functions. The library manager functions are:

- Creating new library files;
- Adding object files to libraries;
- Deleting library object modules;
- Replacing library object modules;
- Copying library object modules;
- Moving library object modules;
- Merging libraries;
- Producing cross-reference-listing files; and
- Checking library object module consistency.

E.5.1 Order of Function Processing

The Lahey Library Manager processes all forms of user input in the following order:

1. After checking for library consistency, the library manager scans the user input to determine whether to create or modify a library and what operations to perform. LM then creates its own set of instructions, similar to a response file, which processes commands in the following order:

2. All deletion commands are processed first. Instead of immediately deleting the specified object modules, LM first marks the object modules to delete. Then it creates a new library file and copies only the unmarked object modules to the new library.

The old library is preserved with the same name and the extension ".BAK". This method eliminates the risk of deleting object modules from a library that may be needed later.

3. All move commands are processed next. The library manager preserves the original library with the same filename and the extension ".BAK".

4. All other commands (additions, replacements and copies) are then processed in the order they were input. LM preserves the original library with the same filename and the extension ".BAK".

NOTE: This process of duplicating the existing library prior to performing any deletions, moves, additions, replacements or copies is for the user's protection. If the library manager operations are terminated in the middle of a process, the original library is not lost. However, enough disk space must be available to accommodate both libraries.

5. LM then assembles the data required to create the library index and a cross-reference-listing file. The library index is used by the linker to search the library for the object modules required in a program, rather than having to search the entire library sequentially at link time. The cross-reference-listing file contains a list of the library's public symbols and the object module names where these symbols are defined. This listing file is only created if the library manager is explicitly directed to do so, even though the listing data is automatically assembled.

E.5.2 Creating New Library Files

In general, there are two ways to create a new library. The first method involves using the command line input method to specify the name of a non-existent library in the old library name field or in the new library name field. The second method is to specify a new library name in response to the **Library name:** or the **Output library name:** prompt. Creating libraries using response files is identical to the prompt response methods described in the Method 2 examples below.

Method 1

Create a new library by entering a non-existent library name in either the old library name field or the new library name field of the LM Command Line along with any options, commands or list filenames followed by a semicolon and a carriage return. If the library name is entered in the old library name field, you must use the Add Modules Command to specify the contents of the new library. If the library name is entered in the new library name field, you may either use the Add Modules Command to specify the library's contents or specify an existing library in the old library name field.

Method 1 Examples:

```
LM <libname>/PA:32+objfile1+objfile2;
```

This creates the **libname** library containing the two specified object file's contents and having a 32-bit page size.

```
LM <oldlib>+objfile1+objfile2,listfile,<newlib>;
```

This creates the **newlib** library with the contents of the **oldlib** library plus the contents of the two specified object files. The library manager also creates the **listfile** cross-reference listing.

```
LM <oldlib>,,<newlib>;
```

This creates the **newlib** library with the contents of the **oldlib** library.

Method 2

Create a new library by entering the LM command and then a non-existent library name in response to the **Library name:** prompt. The library manager supplies the default extension ".LIB", if an extension is not specified. Do not use the name of an existing library. If you do, the library manager will modify the existing library file. When you enter a library name that does not exist, the library manager prints the following text on the screen:

Library file does not exist. Create a new one (Y/N)?

Enter a **Y** to create the new library, or an **N** to terminate the process. If **Y** is selected, LM displays the rest of the input prompts. If **N** is selected, LM terminates operations and returns the DOS prompt.

Use the /PAGESIZE Option Switch when a library is created to specify a library object module page size other than the default of 16 bytes. This will be necessary if the library to be created is to be larger than 1MB.

Answer the **Operations desired:** prompt with the Add Modules Command before the libraries and/or object filenames you want the new library to contain. Then enter a carriage return instead of a library name in response to the **Output library name:** prompt. LM creates a new library with the name you input in response to the **Library name:** prompt and the contents of the libraries and/or object files you specified with the Add Modules Command. If you are adding the contents of a library, you must use the ".LIB" extension.

You can also create a new library by entering an existent library name in response to the **Library name:** prompt. Enter any of the LM commands in response to the **Operations desired:** prompt. Enter a carriage return in response to the **List filename:** prompt. Then enter a new library name in response to the **Output library name:** prompt. LM will create a new library with the contents of the library specified in the **Library name:** prompt but with the name you input in the **Output library name:** prompt. If you specified any LM commands, the new library will reflect the changes you indicated with the LM commands.

The contents of another library can also be added by first extracting the desired object modules from the source library using the /EXTRACTALL Option Switch and then adding the desired extracted object files to the new library.

Method 2 Examples:

LM<CR>

The following prompts appear on the screen. Answer these prompts as shown below:

Library name: libname<CR>

Operations desired: +objfile1+objfile2<CR>

List filename: <CR>

Output library name:<CR>

This example creates the library **libname** containing the contents of the two specified object files. The **libname** specified must not already exist unless you intend to overwrite it. The **libname** does not need an extension if ".LIB" is acceptable.

Use the /PAGESIZE Option Switch when a library is created to specify a library object module page size other than the default of 16 bytes. This will be necessary if the library to be created is to be larger than 1MB.

Entering the name of an existing library preceded by the + symbol and followed by the ".LIB" extension in response to the **Operations desired:** prompt, creates a new library by appending the existing library's modules to the end of the library specified in response to the **Library name:** prompt. The new library contains the contents of the libraries specified at the **Library name:** and the **Operations desired:** prompts, but retains the old library name.

LM<CR>

The following prompts appear on the screen. Answer these prompts as shown below:

Library name:oldlib<CR>

Operations desired: +lib2.lib<CR>

List filename: LPT1<CR>

Output library name: newlib<CR>

This example results in the contents of the **lib2.lib** library being appended to the end of the **oldlib** library and the creation of the **newlib** library. The ".LIB" extension is required input. The listing file is sent to the printer on the LPT1 printer port.

E.5.3 Changing Library Contents

Library files are changed in three ways:

1. By specifying the library to change in the old library name field of the LM Command Line, followed by any LM commands, followed by inputs or defaults for the remaining fields, a semicolon and a carriage return; or

2. By specifying the library to change in response to the **Library name:** prompt and the operations desired in response to the **Operations desired:** prompt, followed by inputs or defaults for the remaining prompts; or

3. By creating a response file containing the name of the library to change, all of the desired operations, the list filename and the output library name.

If a new library name is not specified, the library manager copies the contents of the old library to a library with the same name and the ".BAK" extension and then makes the requested changes to the old library. If a new library name is specified, the requested changes are made to the new library.

E.5.4 Adding Libraries and Object Files to Libraries

Libraries and object files are added to existing libraries by using the addition symbol (+) in front of each library or object filename(s) to be added, in one of three ways:

1. By specifying the library to change in the old library name field of the LM Command Line, followed by the addition symbol (+) in front of each library or object filename(s) to add, followed by inputs or defaults for the remaining fields, a semicolon and a carriage return; or

2. By specifying the library to change in response to the **Library name:** prompt and the addition symbol (+) in front of each library or object filename(s) to add in response to the **Operations desired:** prompt, followed by inputs or defaults for the remaining prompts; or

3. By creating a response file containing the name of the library to change, the desired addition operations, the list filename and the output library name.

You must use the ".LIB" extension for the library to be added. If you do not use the ".LIB" extension, LM will not find the library. If the library or object filename to be added has a drive letter, pathname or extension, the library manager removes these before adding the object file to the end of the library.

Add Object Modules Examples:

```
LM <old lib>+libname2.lib+object.obj;
```

Adds the library **libname2.lib** and the object file **object.obj** and makes it the last object module in the **oldlib** library specified in the old library name field. The ".LIB" extension is required input.

```
LM <old lib>+object1.obj+object2.obj;
```

Adds the two object files (**object1.obj** and **object2.obj**) to the end of the **oldlib** library specified in the old library name field. The use of this command causes the library manager to use the defaults for all non-specified command line input fields.

LM<CR>

Library name: libname1<CR>

Operations desired: +libname2.lib<CR>

List filename: con<CR>

Output library name: libname3<CR>

This is an example of prompt response input that results in the contents of the **libname2.lib** library being added to the **libname1** library to form the new library **libname3**. The cross-reference listing output is sent to the screen using the filename, **con**. The <CR>s in the example refer to carriage returns. Response file input would be identical.

E.5.5 Deleting Library Object Modules

Delete library object modules by preceding each object module name(s) to be deleted with a subtraction symbol (–), in one of three ways:

1. By specifying the library to change in the old library name field of the LM Command Line, followed by the subtraction symbol (–) in front of each object module name(s) to be deleted, followed by inputs or defaults for the remaining fields, a semicolon and a carriage return; or

2. By specifying the library to change in response to the **Library name:** prompt and the subtraction symbol (–) in front of each object module name(s) to be deleted in response to the **Operations desired:** prompt, followed by inputs or defaults for the remaining prompts; or

3. By creating a response file containing the name of the library to change, all of the desired object module deletion operations, the list filename and the output library name.

If a new library name is not specified, the library manager copies the contents of the old library to a library with the same name and the ".BAK" extension and then makes the requested changes to the old library. If a new library name is specified, the requested changes are made to the new library.

Object module names cannot have drive letters, pathnames or extensions, and any attempts to specify these will result in an error message.

Delete Object Modules Examples:

LM <oldlib>–object1;

Deletes the object module **object1** from the **oldlib** library specified in the old library name field. The semicolon at the end of this command causes the library manager to use the defaults for all non-specified command line input fields.

LM<CR>

Library name: libname1<CR>

Operations desired:–module2<CR>

List filename: con<CR>

Output library name: libname3<CR>

This is an example of prompt response input that results in the **module2** module being deleted from the **libname1** library to form the new library **libname3**. The cross-reference-listing output is sent to the screen using the filename **con**. The <CR>s in the example refer to carriage returns. Response file input would be identical.

E.5.6 Replacing Libraries and Object Modules

Replace object modules using the replace command symbol (–+) in front of each object module name(s) to be replaced, in one of three ways:

1. By specifying the library to change in the old library name field of the LM Command Line, followed by the replace symbol (–+) in front of each object module name(s) to be replaced, followed by inputs or defaults for the remaining fields, a semicolon and a carriage return; or

2. By specifying the library to change in response to the **Library name:** prompt and the replace symbol (–+) in front of each object module name(s) to be replaced in response to the **Operations desired:** prompt, followed by inputs or defaults for the remaining prompts; or

3. By creating a response file containing the name of the library to change, all of the desired object module replacement operations, the list filename and the output library name.

When object module replacement is requested, the library manager deletes the object module(s) specified from the library and replaces them with object files of the same name(s) as the deleted or object module(s). The libraries or object files specified to replace those deleted must exist before this operation. The ".OBJ" extension is assumed for the object files that replace the library object modules unless another extension is specified.

If a new library name is not specified, the library manager writes the contents of the old library to a library with the same name and the ".BAK" extension and then makes the requested changes to the old library. If a new library name is specified, the requested changes are made to the new library.

Object module names cannot have drive letters, pathnames or extensions and any attempts to specify these will result in an error message.

Replace Object Modules Examples:

LM <old library name>–+object1–+object2;

Replaces the **object1** object module in the library specified in the old library name field with the **object1.obj** object file and the **object2** object module with an object file of the same name. The semicolon at the end of this command causes the library manager to use the defaults for all non-specified command line input fields.

LM<CR>

Library name: libname1<CR>

Operations desired:–+object1<CR>

List filename: con<CR>

Output library name: libname3<CR>

This is an example of prompt response input that results in the contents of the **object1** module being replaced by a new version of the **object1** object file in the **libname1** library to form the new library **libname3**. The cross-reference-listing output is sent to the screen using the filename **con**. The <CR>s in the example refer to carriage returns. Response file input would be identical.

E.5.7 Copying Library Object Modules

Library object modules are copied using the copy command symbol (*) in front of each object module name(s) to be copied, in one of three ways:

1. By specifying the library to change in the old library name field of the LM Command Line, followed by the copy symbol (*) in front of each object module name(s) to be copied, followed by inputs or defaults for the remaining fields, a semicolon and a carriage return; or

2. By specifying the library to change in response to the **Library name:** prompt and the copy symbol (*) in front of each object module name(s) to be copied in response to the **Operations desired:** prompt, followed by inputs or defaults for the remaining prompts; or

3. By creating a response file containing the name of the library to change, all of the desired object module copy operations, the list filename and the output library name.

When object module copying is requested, the library manager copies the object module(s) specified from the library and writes them to object file(s) with the same name(s) as the copied object module(s). The ".OBJ" extension is assumed for the object files that result from the copy operation. Library object modules cannot have drive letters, pathnames or extensions and any attempts to specify these will result in an error message. When the object module is copied to an object file, the library manager automatically adds the drive letter and pathname of the current working directory to the new object files. To move or rename the new object files, use the DOS Copy or Rename Commands to give the new object files new names, drive letters, pathnames and/or extensions.

Copy Object Modules Examples:

```
LM <old lib>*object1*object2;
```

This example copies the **object1** and **object2** object modules to object files named **object1.obj** and **object2.obj** in the current directory. The semicolon at the end of this command causes the library manager to use the defaults for all non-specified command line input fields.

LM<CR>

Library name: libname1<CR>

Operations desired: *module2<CR>

List filename: <CR>

Output library name:<CR>

This is an example of prompt response input that results in the **module2** module being copied from the **libname1** library to a **module2.obj** object file. No cross-reference-listing output file and no new library file are created. The <CR>s in the example refer to carriage returns. Response file input would be identical.

E.5.8 Moving Library Object Modules

The move command symbol (–*) relocates specified library object modules to object files with the same name(s) as the library object modules. This command functions almost like the copy command, except the specified object modules are deleted from the library. Use the move command in one of three ways:

1. By specifying the library to change in the old library name field of the LM Command Line, followed by the move symbol (–*) in front of each object module name(s) to be moved, followed by inputs or defaults for the remaining fields, a semicolon and a carriage return; or

2. By specifying the library to change in response to the **Library name:** prompt and the move symbol (–*) in front of each object module name(s) to be moved in response to the **Operations desired:** prompt, followed by inputs or defaults for the remaining prompts; or

3. By creating a response file containing the name of the library to change, all of the desired object module move operations, the list filename and the output library name.

When the move command is requested, the library manager writes the object modules to object files with the same name(s) as the moved object module(s) and deletes these modules from the library. The ".OBJ" extension is assumed for the object files that result from the move operation. Library object modules cannot have drive letters, pathnames or extensions and any attempts to specify these will result in an error message. When the object module is moved to an object file, the library manager automatically adds the drive letter and pathname of the current working directory to the new object files. To move or rename the new object files, use the DOS Copy or Rename Commands to give the new object files new names, drive letters, pathnames and/or extensions.

Move Object Modules Examples:

LM <old lib> –*module1*module2, ,newlib;

This example performs a series of actions. First, the object module **module1** is moved from the **oldlib** library to an object file called **module1** and the **module1** object module is deleted from the **oldlib** library. Second, the object module **module2** is copied from the **oldlib** library to an object file named **module2.obj**. The original **module2** remains in the original library. Third, the two commas following the **module2** in the example enforce the default of no cross-reference-listing file being created. Fourth, a new library is created with the name, **newlib**, specified in the new library name field. This new library contains all of the contents of the original library except the **module1** object module, which was deleted when it was moved to create an object file of the same name. The **oldlib** library retains all of its original contents.

LM<CR>

Library name: libname1<CR>

Operations desired: –*module2<CR>

List filename: <CR>

Output library name:<CR>

This is an example of prompt response input that results in the **module2** module being moved from the **libname1** library to a **module2.obj** object file and deleted from the **libname1** library. No cross-reference-listing output file and no new library file are created. The <CR>s in the example refer to carriage returns. Response file input would be identical.

E.5.9 Merging Libraries

There are two ways to merge libraries. The first method is to use the Add Modules Command to combine the contents of two existing libraries. The second method is a two-step process using the /EXTRACTALL Option Switch. The first step is to specify the library containing the required object modules on the LM Command Line, followed by the /EXTRACTALL Option Switch. This process copies all of that library's object modules to corresponding object files with the same names. The second step is to use the Add Modules Command (see Section E.1.6) to add these object files to another library as object modules. Be sure to specify a new library name in the new library name field of the LM Command Line so that the original library is preserved.

Merging Libraries Method 1 Example:

LM <libname1> +libname2.lib;

adds the **libname2.lib** library to the **libname1** library. The ".LIB" extension is required input. The semicolon at the end of this command causes the library manager to use the defaults for all non-specified command line input fields.

LM<CR>

Library name: libname1<CR>

Operations desired: +libname2.lib<CR>

List filename: LPT1<CR>

Output library name:libname3<CR>

This is an example of prompt response input that results in the **libname2.lib** library being added to the **libname1** library to create the **libname3** library. The ".LIB" extension is required input. The cross-reference-listing output file is sent to the printer on the LPT1 printer port. The <CR>s in the example refer to carriage returns. Response file input would be identical.

Merging Libraries Method 2 Example:

```
LM <libname1>/EX;
```

extracts all object modules from the **libname1** library and writes them to object files with the same names. Then use the Add Modules Command as shown below:

```
LM <libname2>+object1+object2+object3;
```

to add the **object1, object2** and **object3** object modules, that were extracted from the **libname1** library, to the **libname2** library. The semicolon at the end of this command causes the library manager to use the defaults for all non-specified command line input fields.

E.5.10 Cross-Reference Listing File Output

Cross-reference-listing files create two lists. The first list shows all public symbols for each library object module in alphabetical order. The second lists library object modules in alphabetical order. Listing files can be specified in one of three ways:

1. By specifying the library in the old library name field of the LM Command Line, followed by either commands or commas to force a default in the commands field, followed by the name of the listing file (or output device) in the listing file field, followed by input or commas to force a default for the new library name field, a semicolon and a carriage return; or

2. By specifying the library in response to the **Library name:** prompt, followed by commands or a carriage return in response to the **Operations desired:** prompt, followed by the listing filename (or output

device) in response to the **List filename:** prompt, followed by an output library name or a carriage return in response to the **Output library name:** prompt, or

3. By creating a response file containing the name of the library, any desired operations, the name of the desired listing file (or output device) and the name of an output file or a carriage return to force the default of none.

If no listing file is explicitly requested, the library manager will suppress creation of a listing file. There are no default filenames or extensions for cross-reference-listing files.

Cross-Reference-Listing Examples:

```
LM <old library name>,listfile.pub;
```

This example creates the **listfile.pub** cross-reference-listing file for the library specified in the old library name field and does not perform any other function on the library. The semicolon at the end of this command causes the library manager to use the defaults for all non-specified command line input fields.

The next example:

```
LM <old library name>,con;
```

creates a cross-reference-listing file for the library specified in the old library name field and sends the file output to the screen. The semicolon at the end of this command causes the library manager to use the defaults for all non-specified command line input fields. The next example:

```
LM <old library name>,LPT1;
```

creates a cross-reference-listing file for the library specified in the old library name field and sends the file output to a printer on the LPT1 printer port. The next example:

```
LM <oldlib>+object1,listfile.pub,newlib;
```

adds the **object1** object file to the **oldlib** library, generates the **listfile.pub** cross-reference-listing file and creates the **newlib** library.

The next example:

LM<CR>

Library name: libname1<CR>

Operations desired: +libname2.lib<CR>

List filename: LPT1<CR>

Output library name:libname3<CR>

is an example of prompt response input that results in the **libname2.lib** library being added to the **libname1** library to create the **libname3** library. The cross-reference-listing output file is sent to the printer on the LPT1 printer port. The <CR>s in the example refer to carriage returns. Response file input would be identical.

E.5.11 Checking Library Object Module Consistency

Consistency checking ensures that all library object modules are functional. If no errors are found in the object modules, the library manager issues no message and returns the DOS prompt. If errors are found, the library manager issues error messages and then returns the DOS prompt.

Consistency checks are performed in one of three ways:

1. By specifying the library to check in the old library name field of the LM Command Line followed by a semicolon and a carriage return; or

2. By specifying the library name to check in response to the **Library name:** prompt, followed by three carriage returns; or

3. By creating a response file containing the name of the library to check, followed by three carriage returns.

The library manager routinely performs consistency checks on all object files added to libraries. It is not usually necessary to check libraries created with the library manager, although older libraries created with other products should be checked.

Consistency Checking Examples:

LM <oldlib>;

checks all of the object modules in the **oldlib** library specified in the old library name field for consistency and outputs error messages if any object modules are flawed. The next example:

LM <oldlib>,listfile.pub;

checks all object modules in the **oldlib** library specified in the old library name field for consistency and and then creates the **listfile.pub** cross-reference-listing file.

LM<CR>

Library name: libname1;<CR>

Operations desired: <CR>

List filename: <CR>

Output library name:<CR>

This is an example of prompt response input that results in the **libname1** library being checked for module consistency. The <CR>s in the example refer to carriage returns. Response file input would be identical.

E.6 Library Manager Error Messages

This section lists the messages produced when the Lahey Library Manager encounters an error condition. Messages are grouped into four classes: ABORT, FATAL, WARNING and PROMPTS. The errors listed below are followed by explanations and suggested solutions.

Error: **'AA'('BB'): symbol redefinition in file 'CC'('DD') ignored**

Type: WARNING

Meaning: The public symbol "AA" which was gotten from module "DD" in the object module file "CC" already exists in the library in the module "BB". This is only a warning. The library's index directory will continue to point to the symbol that is located in module "BB".

Error: **Cannot create file <filename>**

Type: FATAL

Meaning: A DOS file create failed for file <filename>. This can happen due to two reasons:

1. Disk directory is full;

2. File <filename> already exists and cannot be overwritten due to its access permissions.

Error: **In build TOC — couldn't fit symbol <symbol>**

Type: FATAL

Meaning: This is an LM internal error. If you get this message, call Lahey Computer Systems at: 702 831-2500.

Error: **Invalid command specified**

Type: FATAL

Meaning: The given command was not ADD, DELETE, REPLACE, COPY or MOVE.

Error: **Invalid file specification**

Type: FATAL

Meaning: The given filename does not conform to the DOS file naming convention.

Error: **Invalid object module format in file <filename>**

Type: FATAL

Meaning: The first record type of any specified object module must be a THEADR type. This error occurs if this is not the case.

Error: **Invalid pagesize specified**

Type: FATAL

Meaning: Page size must be in integral powers of 2 from 16 to 32768, inclusive. The specified page size was not valid.

Error: **Invalid record type <type> in file <filename> near <offset>**

Type: FATAL

Meaning: Near offset <offset> in file <filename>, an unknown object module record type was encountered. This usually means that the object module or file is corrupted.

Error: **Library does not exist. Create a new one (Y/N)?**

Type: PROMPT

Meaning: The library manager cannot find the file specified in the old library name field or in response to the **Library name:** prompt. Enter a **Y** to create a new library with the name specified or an **N** to return to the LM prompt.

Error: **Library was just created — deletion of <name> ignored**

Error: **Library was just created — copy of <name> ignored**

Error: **Library was just created — extraction of <name> ignored**

Type: WARNING

Meaning: When a new library is created, the DELETE and COPY commands and the /EXTRACTALL Option Switch are ignored.

Error: **LM: Internal error — ran out of prime numbers**

Type: FATAL

Meaning: This is an LM internal error. If you get this message, call Lahey Computer Systems at: 702 831-2500.

Error: **Memory allocation failed**

Type: FATAL

Meaning: The computer's free memory pool has been exhausted. Try reducing the number of object modules being specified.

Error: **Missing library name**

Type: FATAL

Meaning: A library name must always be specified.

Error: **Module <name> already exists in library — ADD command ignored**

Type: WARNING

Meaning: A module with the same name already exists in the library, so this command was ignored.

Error: **Module <name> not found**

Type: WARNING

Meaning: A DELETE, REPLACE, MOVE or COPY of module <name> failed because the module was not found in the library.

Error: **Too many object modules specified**

Type: FATAL

Meaning: LM can support up to 512 object modules at one time. If this limit is exceeded, then this error will occur. Try reducing the number of object modules.

Error: **Too many modules — table overflow**

Error: **Too many public symbols — table overflow**

Type: FATAL

Meaning: These severe errors mean that LM's limitations on the number of modules or public symbols have been exceeded. Contact LCS with details.

Error: **Unable to open file <filename>**

Type: FATAL

Meaning: Object file <filename> could not be found.

Error: **Unexpected end-of-file in file <filename>**

Type: FATAL

Meaning: This usually means that file <filename> is corrupted.

Error: **Write error — disk full**

Type: FATAL

Meaning: Insufficient space left on disk drive.

F LAHEY OPTLINK LINKER BY SLR SYSTEMS

This appendix describes using the Lahey OPTLINK Linker to create executable program files from assembled or compiled object language modules.

The appendix does not provide a tutorial for learning how to use linkers or link-editing programs, but it does provide information adequate to permit experienced programmers to teach themselves such techniques.

Lahey Computer Systems only warrants and provides support for the features and options found in this appendix that are applicable to FORTRAN programming. Lahey OPTLINK features which apply to other languages and/or are used to create ".SYS" or ".COM" files must be used at the user's discretion.

F.0.1 Typographical Conventions

Unique Lahey OPTLINK Linker design features appear in bold, underlined text with the feature explained in indented text blocks. For example:

OPTLINK Feature Lahey OPTLINK permits the user to configure most aspects of error detection.

F.1 Introduction

Lahey OPTLINK is a superfast program linker designed for use by professional programmers to replace DOS LINK and other compatible linkage editors.

Lahey OPTLINK works with modules in INTEL OMF Format (.OBJ files), including those generated by standard assemblers and compilers. The program files that it produces are fully compatible with those produced by LINK, PLINK, and TLINK.

F.1.1 Purpose of Lahey OPTLINK

Lahey OPTLINK is useful for a number of purposes:

- It can create executable program files in ".EXE" format from object modules produced by both OPTASM or compatible assemblers, and by high-level languages which produce standard OMF ".OBJ" files.

- It can also produce executable program files in ".COM" format, or in the zero-relative ".SYS" format used for installable device drivers.

NOTE: References in this manual to DOS refer to IBM PC-DOS and Microsoft MS-DOS. Similarly, references to "LINK" or "linker" refer to the IBM PC-DOS Linker, the Microsoft MS-DOS Linker, the Phoenix Technology PLINK Linker, or Borland International's TLINK Linker. References to "LIB" refer to the Lahey Library Manager, OPTLIB, the IBM PC-DOS Librarian, and the Microsoft MS-DOS Librarian.

F.1.2 Features of Lahey OPTLINK

Although compatibility with existing linkers was a prime objective in the design of Lahey OPTLINK, the product sets itself apart from other linkers with a number of exclusive features.

Speed
Lahey OPTLINK executes faster than any other linker for standard MS-DOS files.

Simplicity
Lahey OPTLINK directly generates a ".COM" file when the output extension is set to ".COM", or a ".SYS" file (ORG 0) when the output extension is set to ".SYS". No "EXE2BIN" utility need be used.

Use of LIM Memory
The virtual memory system in Lahey OPTLINK uses LIM (Lotus Intel Microsoft Standard) memory if it is available.

Cross Referencing
Complete global cross-reference generation capability is provided.

User Configurable
Lahey OPTLINK may be configured to one's own preferences at installation time, and the configuration may be changed whenever desired.

Error Detection
Other linkers ignore the fact that some segments are absolute (like screen segment at B800h, or public constants, which appear as 'segment at 0' with an offset) and some are relocatable. Lahey OPTLINK permits such errors to be detected, or ignored, depending on user configuration choices.

Error Treatment

Checksum errors in any input module can be detected, or ignored, as desired. Most other linkers always ignore such errors since some popular languages do not generate checksums in their output.

Environment Use

Lahey OPTLINK can use dedicated environment variables, which add to the configurability by storing commands for default use.

F.1.3 Getting Started

If you are already familiar with the use of other linkers, you can use Lahey OPTLINK in exactly the same way as MS-LINK. As shipped, all defaults and configuration options are set for complete compatibility. To begin using the Lahey OPTLINK Linker, type:

OPTLINK /HE

Where:

"OPTLINK" is the linker executable; and

"/HE" invokes the linker help option to display parsed switches.

F.2 Configuring Lahey OPTLINK

The "OPTFIG.EXE" Configuration Program tailors default operating parameters of the linker to a particular environment. Execution of the configuration program is optional, since all of the parameters have preset values. Taking the step of initially configuring these parameters can relieve the programmer from having to override default values that conflict with common operating modes.

F.2.1 Using the Configuration Program

The configuration program, "OPTFIG.EXE", requires two files: "OPTLINK.EXE" and "OPTLINK.FIG". To configure Lahey OPTLINK, type:

OPTFIG OPTLINK

If for any reason the release levels of the "OPTLINK.EXE" and "OPTLINK.FIG" files fail to match, the response will be:

Not OPTLINK Release x.xx

In such a case, OPTFIG will halt. Normally, however, the program responds:

OPTLINK Configuration Utility (C) 1989 SLR Systems

Press **<CTRL-C>** to abort, **<ESC>** to save & exit, **<RETURN>** or **<ENTER>** for the next option, and **<SHIFT–6>** for the previous option, or the appropriate arrow keys. Use **?** for help.

The program then prompts the user for a series of configuration options.

Operation of the program is controlled by the following keyboard responses:

ENTER (or **RETURN**)	display next option
(DOWN–ARROW)	display next option
(UP–ARROW)	display previous option
^ (SHIFT-6)	display previous option
HOME	display first option
END	display last option
?	help for any OPTFIG question
ESC	save configuration and exit
Ctrl-C	abort configuration

The next section lists the configuration options with explanations in the order they are displayed by the program.

After configuration has been completed, the "OPTFIG.EXE" and "OPTLINK.FIG" Files may be deleted from the working copy of the linker disk. They are not required for execution of Lahey OPTLINK.

F.2.2 Lahey OPTLINK Configuration Options

The "OPTFIG.EXE" Configuration Program displays each configuration option followed by its present setting (shown in parentheses). This section lists the options, their descriptions, and the settings of the options at the time Lahey OPTLINK is delivered. The letters "Y" and "N" in an option display designate YES and NO.

In many cases, configuration settings may be overridden either by an environment variable setting or by associated command options. The precedence order is:

(highest)
1 command-line options
2 environment variable command options
3 configuration settings
(lowest)

[/MAP] Create .MAP file (N)

Y Lahey OPTLINK will automatically create a .MAP file suitable for use with debugging programs, or for interpreting a machine code dump of the program. This file contains all segment definitions and public symbols (sorted numerically), and will have the same name as the ".EXE" file but with the ".MAP" extension.

N No ".MAP" file is created automatically. One can still be requested via the command-line syntax, or in interactive operation.

[/LINENUMBERS] Put Linenumber Info In MAP file (N)

Y If a ".MAP" file is being created, due either to the preceding configuration factor or to any option switch or prompt response, Lahey OPTLINK adds line-number information to the file (provided, of course, that such information is present in the input ".OBJ" modules). This makes possible source-level debugging when using debuggers which support such action.

N No line number information is included when a ".MAP" file is produced.

[/XREF] Create .MAP file with cross reference (N)

Y If a ".MAP" file is being created, due either to the preceding configuration factor or to any option switch or prompt response, Lahey OPTLINK appends a complete global cross-reference listing to the file.

N No global cross-reference listing is produced.

[/IGNORECASE] Do case-insensitive link (Y)

Y All symbol name references and definitions in input files (both ".OBJ" and ".LIB" files) are mapped into uppercase during lookups. Thus "InBuffer", "inbuffer", and "INBUFFER" all refer to the same symbol with this setting.

N No case mapping occurs; symbols must match in case in addition to everything else, in order to match. Thus "InBuffer", "inbuffer", and "INBUFFER" refer to three different symbols with this setting. This setting is required for operation with most C compilers.

[/CHECKSUM] Detect .OBJ & .LIB checksum errors (N)

Y Checksums are calculated for all data read from input files, and compared to the checksum stored in the file itself. If the two do not match, a warning is issued (or, if the following switch is also set, a fatal error is declared). Not all assemblers and compilers store correct checksums in their output files.

N No checksums are calculated or checked.

[/CHECKABORT] Abort on BAD CHECKSUM error (N)

Y If the previous switch is also set, enabling checksum calculation, detection of a checksum mismatch results in a fatal error. If checksums are not enabled this switch has no effect.

N If the previous switch is also set, enabling checksum calculation, detection of a checksum mismatch results in a warning but linking continues. If checksums are not enabled, this switch has no effect.

[/CHECKEXE] Calculate Checksum on Output EXE (N)

Y Lahey OPTLINK generates ".EXE" checksum word (like MS-LINK).

N Lahey OPTLINK does not generate ".EXE" checksum word (like TLINK).

[/ECHOINDIRECT] Echo Indirect File to Screen (N)

Y Lahey OPTLINK echoes indirect command files to the console.

N Lahey OPTLINK does not echo indirect command files to the console.

[/DOSSEG] Select DOS Segment Ordering (N)

Y Segments in the ".EXE" file are arranged in DOS sequence, with all code first followed by data.

N Classes in the ".EXE" file are arranged in the sequence in which they are first encountered in the ".OBJ" modules. Segments are arranged within classes similarly, in the sequence in which they are first encountered.

[/DEFAULTLIBRARYSEARCH] Utilize .LIB search info from OBJ files (Y)

Y Lahey OPTLINK honors a comment record in an ".OBJ" file requesting a library search.

N Lahey OPTLINK ignores a comment record in an ".OBJ" file requesting a library search.

[/GROUPASSOCIATION] Utilize GROUP info from OBJ files (Y)

Y Lahey OPTLINK uses all GROUP information found in the ".OBJ" files.

N Lahey OPTLINK makes no use of the GROUP information in the ".OBJ" files.

[/CODEVIEW] Put debugging info at end of EXE (N)

Y CodeView-format debugging information is appended to the ".EXE" file in same manner as Microsoft's LINK.

N No debugging data is appended to the ".EXE" file.

NOTE: Lahey FORTRAN-generated object files do not support CodeView.

[/PAUSE] Wait for disk change before Output (N)

Y Lahey OPTLINK pauses to permit disk change before writing output files.

N Lahey OPTLINK writes output files without pausing.

Ignore Absolute/Relocatable Conflicts (for Borland) (Y)

Y Permits use with languages which supply a relocatable frame of 0 (which means 0 from your PSP) as frame from which to access constants defined as "segment at 0" (an absolute rather than relocatable definition). Technically, this is illegal under the standard rules for ".OBJ" files, but since some of the most popular languages available (notably, Borland's Turbo-C V.2.0) do so, Lahey OPTLINK provides this switch so that you can configure to accept such usage.

N Generates a fatal error when a relocatable reference is made to an absolute segment or vice versa, in strict accordance with standard rules for ".OBJ" files.

Put GROUP-Based SS:SP in EXE header (Y)

In simplified-segmentation mode, and in many other cases, the stack is a part of DGROUP, or some group in general. When other linkers set up the stack pointer in the ".EXE", they always put the physical stack segment and offset, not the group base and offset-from-group. Because of this, you must reset the stack and stack segment manually, if you want to use SS=DGROUP (or whatever your group is). This configuration option permits the linker to do it correctly the first time.

Y Write SS:SP in Group-based format in the ".EXE" header.

N Write SS:SP as physical segment:offset, just as other linkers do.

After the final option selection is entered, CONFIG displays the following message:

---- End of options, press ESC to save and exit. ----

F.3 Using Lahey OPTLINK

Lahey OPTLINK may be run in any of three ways:

- the first method is to enter all required input in response to prompts issued by the program (interactive operation),
- the second is to provide all the input as a series of parameters on the command line (command-line control), and
- the third is to provide the prompt responses in a separate file (indirect file operation) and to pass that file's name to the program on the command line.

F.3.1 Interactive Operation

The simplest way to use Lahey OPTLINK is to type:

OPTLINK

The program will then issue a number of prompts. The exact number is determined, at least in part, by the information entered in response to the previous prompt. This interaction between the input and the program's next prompt gives the mode its name.

OPTLINK Prompts

The first prompt issued will always be the Name Prompt:

OBJ Files: (NUL.OBJ):

The response to this prompt must be one or more file names, separated by spaces or by "+" characters. Lahey OPTLINK will assume the default file extension is ".OBJ" if none is entered. The file or files must exist. If a "+" character appears at the end of the line, Lahey OPTLINK will repeat this prompt and accept additional names.

In response to this prompt, and in all other places where an ".OBJ" file name is expected, ambiguous filenames are supported. That is, the "wild-card" characters "?" and "*" are legal for specifying ".OBJ" file names. Unambiguous references override ambiguous ones. That is, you can say:

MY_FIRST * MY_LAST

to have "MY_FIRST" linked first, "MY_LAST" last, and all other ".OBJ" files in that directory in between.

If the reply to the Name Prompt is one of the DOS Reserved Device Names, or simply **ENTER**, the response will be:

Error 13: Bad Input Device

and the operation will abort.

When the input file name has been entered, Lahey OPTLINK responds with its second prompt, which is automatically modified by the first response. It appears as:

Output File: (xxx.EXE):

Where:

"xxx" is the name of the first ".OBJ" file you supplied to the Name Prompt. Although the default extension is ".EXE", as suggested by the prompt, you may enter ".COM" or ".SYS" if you desire to create your program file in either of these non-EXE formats.

Output File Formats The ability to create either ".EXE", ".COM" or ".SYS" format output files automatically is unique to Lahey OPTLINK.

The third prompt appears as:

Map File:(NUL.MAP):

This names the file to which the map listing will be written. Unless the /MAP switch has been set by user configuration, replying with **ENTER** will suppress all MAP output. If the /MAP switch is set, the suggested filename will be the same as that for the output file set by the previous prompt.

If output is to go instead to some other file, or to a device, simply supply the appropriate name. Lahey OPTLINK will assume the default file extension of ".MAP" unless some other is specified. For no extension at all, add a period after the file name.

Supplying a filename to this prompt in order to generate a ".MAP" file, without also using the option switches or configuration choices (/MAP, /XREF, or /LI), yields a file containing only the segment definitions. The /MAP switch adds PUBLIC symbols, /XREF adds a cross reference listing, and /LI adds line number information if present in the input data.

The final prompt names the libraries and search paths to be used. It is:

Libraries and Paths:(NUL.LIB):

The response to this (as to the first prompt) must be one or more filenames or full pathnames, separated by spaces or by "+" characters. Lahey OPTLINK will assume the default file extension is ".LIB" if none is entered. The file or files must exist.

If a "+" character appears at the end of the line, Lahey OPTLINK will repeat this prompt and accept additional names.

F.3.2 Lahey OPTLINK Examples

Several examples of Lahey OPTLINK under interactive operation follow. Operator responses to the prompts appear in **BOLD type** to distinguish them:

C>OPTLINK
Lahey OPTLINK Copyright (C) SLR Systems 1989
Version X.XX
All rights reserved.
OBJ Files: (NUL.OBJ):**curset**
Output File: (CURSET.EXE):
Map File: (NUL.MAP):
Libraries and Paths: (NUL.LIB):

Warning 22: No Stack

The second example illustrates the error response when **ENTER** is pressed after the first prompt:

C>OPTLINK
Lahey OPTLINK Copyright (C) SLR Systems 1989
Version X.XX
All rights reserved.
OBJ Files: (NUL.OBJ):

Error 13: Bad Input Device

F.3.3 Command-line Control

Command-line control of Lahey OPTLINK uses the syntax:

OPTLINK <obj>[,] <out>[,] <map>[,] <lib><;>

Where:

"<obj>" is one or more Input (".OBJ") file names. If multiple names are supplied, they may be joined by "+" characters or by blank spaces. The comma serves to separate the four parts of the command line. If more names are required than are able to fit onto the command line, a "+" immediately before the end of the line can be used to continue the list to the next line.

NOTE: Comma delimiters are required even if <out>, <map>, and <lib> filenames are not supplied in the link command line.

Ambiguous (wildcard) names may be supplied. If specific names are preceded or followed by wildcard references, those modules specifically named will not be included twice. The input list may be followed by specific data, or a semi colon (";") character to indicate that all default choices are to be used for the remaining four parts. If neither is supplied, operation switches to the interactive mode and you are prompted for the remaining inputs.

"<out>" is a single file name specifying the name to be used for the output program file. If no extension is supplied, an ".EXE" file will be generated; if ".COM" is supplied as the extension, a ".COM" file will be created automatically, and if ".SYS" is used as the extension, a binary device-driver format file will be created. If this entry is skipped by supplying two consecutive commas, the default file name, with ".EXE" extension, will be used.

"<map>" is also a single file name and provides the name of the file which will receive all MAP output. It, too, may be defaulted, and follows the same rules as for interactive operation.

"<lib>", like the ".OBJ" entry, can be either a single filename, or a list of files, which are the libraries to be searched in order to resolve any symbols which are not defined in the input ".OBJ" files. The libraries will be searched in the order specified here. The same separators used for the "<obj>" portion apply, and the "<lib>" entry may also be extended past the end of a line by means of the "+" character immediately before the end-of-line. The "<lib>" entry may also be a pathname (with trailing backslash) to a directory containing the libraries to be used.

Library files themselves are searched in the order that they are presented to Lahey OPTLINK. That is, those named by the LIB= variable are searched first for undefined global symbols, followed by those named on the command line, and finally by those referenced by embedded commands in ".OBJ" modules. To locate each library file, if no path is specified, the current directory is searched first, followed by any paths specified in the LIB= variable, and finally any paths named in the command line.

F.3.4 Indirect File Operation

Indirect file operation of Lahey OPTLINK uses the syntax:

OPTLINK @<indirfile>

Where:

"<indirfile>" is the name of an indirect response file. The indirect response file has a number of text lines, one for each prompt that would be issued during interactive operation.

For instance, assume that the indirect file, "LNKRES", contains the following lines (the symbol <CR> represents an empty line containing only a CR/LF pair):

curset
<CR>
<CR>
<CR>

Invoking Lahey OPTLINK for indirect operation with the command line:

OPTLINK @LNKRES

will cause exactly the same sequence of operations as did the first example shown under Interactive Operation, with only the "stack warning" message echoed to the console. The first line in "LNKRES" is the response for the ".OBJ" Files prompt, the next is the response to the Output File prompt, the next is the response to the Map File prompt, and the final line is the **ENTER** response to accept the suggested Library/Path name of NUL.LIB.

F.3.5 Special Filenames

Since Lahey OPTLINK uses standard DOS functions for keyboard input and screen output, device names may be used in place of file names, and input and output may be redirected. The device names are:

Name	Device
AUX	refers to an auxiliary device (usually the same as COM1)
CON	refers to the console (keyboard input or display output)
PRN	refers to the printer (usually the same as LPT1)
NUL	a "null" (nonexistent) file. If given as an input name, returns end-of-file; if given as an output name, no output is generated.

Even when drive designations or extensions are added to these special file names, they remain associated with the listed devices. For example, "A:CON.LST" still refers to the console device and is not the name of a disk file.

To redirect Lahey OPTLINK input and output from and to files, use the standard redirection keys. For example, the following command redirects Lahey OPTLINK output (including all prompts) to file, "OPTLINK.LOG" instead of to the screen:

OPTLINK UTILITY; OPTLINK.LOG

F.3.6 Keyboard Controls

Lahey OPTLINK responds to the standard DOS keyboard controls:

Ctrl-S, Ctrl-NumLock, or Pause	Freezes screen display until another key is pressed.
Ctrl-C or Ctrl-Break	Cancels the program and returns control to DOS.
Ctrl-P, Ctrl-PrtSc, or Print Screen	Activates printing; every screen line is printed after it is displayed. Pressing Ctrl-P a second time deactivates simultaneous displaying and printing.

F.4 Lahey OPTLINK Capabilities

This section describes the functional capabilities of Lahey OPTLINK. The purpose of any linker is to combine separately generated object modules into a single relocatable load module to form an executable program. Such an executable file is identified to DOS by the file type ".EXE" (for executable); in addition to generating ".EXE" files, Lahey OPTLINK can also generate fully bound files in either ".COM" or ".SYS" format. The type of file generated is determined automatically by the file type you provide for the output filename.

Lahey OPTLINK performs a number of sequential actions to accomplish its function. These are partly automatic and partly under your control. They are described in the following paragraphs.

F.4.1 Reading Object Modules

The first action taken by Lahey OPTLINK (after setting all applicable switches for the current run) is to read all ".OBJ" files, in the sequence in which they are specified in the reply to the input prompt, on the command line, or in the indirect file (as applicable). You control the sequence in which files are read by the sequence in which you provide them to Lahey OPTLINK.

In some cases, this sequence is significant. As Lahey OPTLINK reads the files this first time, it collects information about sizes, segments, and symbols for use in the later stages of linking. If a requested object module cannot be found, operation terminates with a fatal error.

F.4.2 Searching Library Modules

After all the object modules named in the input have been read, Lahey OPTLINK then searches through all applicable library (".LIB") modules if any EXTERN symbols remain undefined. The modules first searched are those named by the LIB= variable, followed by those named in the command line or interactive prompt, and finally those named in the input data. Like the object modules, they are searched in the order in which they were named. The first PUBLIC symbol encountered that matches an undefined EXTERN is used and any subsequent occurrences of that symbol as a PUBLIC are ignored, so the sequence in which libraries are searched may have significant impact on a program's operation.

After all named libraries have been searched, or if no libraries are named in the input, the libraries called for by internal records of the object modules (i.e. requested by the translator which generated the object modules) are searched unless this capability has been turned off by configuration or the use of the /NODE (NODEFAULTLIBRARY) switch.

In library searches when no path is specified with the library name, the current default directory is searched first when looking for any specific ".LIB" file. If none is found there, the path(s) listed in the LIB environment variable are used. If a requested library module cannot be found, a warning message is issued but operation continues.

F.4.3 Assigning Segment Addresses

A segment is any continuous sequence of memory locations up to 65536 bytes long. Every segment must begin on a paragraph boundary (an address of which the low four bits are all zero). Every segment referenced in a program is identified by two things: a segment name and a class name. The segment name is assigned by you, if you write in assembler, when you use the SEGMENT/ENDS declarations; if the object module was generated by a high level language, the segment name is assigned by the translator. The class name is also supplied by you, by means of a modifier you may add to the SEGMENT declaration. If given, the class name is enclosed in single quotes, as in:

```
CODESEG   SEGMENT   PUBLIC BYTE 'CODE'
```

In this example, the segment name is CODESEG and the class name is CODE. Lahey OPTLINK combines segments having matching segment and class names based on their combine type, which may be PUBLIC, COMMON, or PRIVATE. PUBLIC segments combine into a single bigger segment; COMMON segments are assigned the same address (i.e. are overlaid onto each other), and PRIVATE segments are not combined at all. Within each program, all segments of the same class are loaded in memory adjacent to each other. The final attribute of a segment is its alignment, which may be BYTE, WORD, DWORD, PARA, or PAGE (corresponding to boundaries at multiples of 1, 2, 4, 16, or 256, respectively. If no alignment is specified, PARA is used.

Lahey OPTLINK assigns segment sizes as each segment is encountered while reading the object modules and library modules. The starting address for each segment depends on its specified alignment, its class name, and the lengths of all preceding segments. For each segment, a frame number is generated which consists of the address of the first paragraph that will contain any of that segment.

Segments may be, and often are, collected into groups by language translators.

The difference between segments in a group and those which are not is that in a group, the applicable segment register is not changed when moving from one segment to another within the group. If groups are not used, code to change the segment register is required whenever the segment changes. Like segments themselves, the size of any group may not exceed 65536 bytes (the maximum that can be addressed without changing segment register contents).

F.4.4 Collecting Relocation Information

When all the segments, groups, combine types, and classes have been sorted out and processed appropriately, addresses are assigned to all the segments. With all addresses known, relocation tables can be generated (internal to Lahey OPTLINK) which are then used to reconcile address references.

F.4.5 Assigning Public Addresses

All public symbols within any object module are identified as such, and only such public symbols can be addressed from outside the module. Lahey OPTLINK keeps track of the segment and offset values for each public symbol encountered, both while reading the object modules and when a library search locates a module that contains a needed extern symbol declared to be public.

F.4.6 Reconciling Address References (Fix-Ups)

Every standard ".OBJ" module normally contains several records devoted to relocation information. These records, identified in the OMF documentation as type FIXUPP, are universally called "fix-ups". They are processed after all segment addresses and symbol references are known, to perform final reconciliation of cross-module references.

F.4.7 Writing Output File(s)

When all address references have been reconciled, Lahey OPTLINK writes the output executable file (and any additional report files if requested). The file is written using the name (and path, if specified) requested in the input data. If none was

specified, the default output filename is the name of the first object module read, and the default file type is ".EXE"; the file will be written in the current directory.

In the normal (default) case, a standard DOS ".EXE" file containing a relocation header followed by the program itself is written. If the output file type was specified as either ".COM" or ".SYS", however, only the memory image of the file itself, fully bound for loading at a fixed offset of 0100h (COM type) or 0000h (SYS type) is produced.

F.5 Lahey OPTLINK Options

This section lists all options recognized by Lahey OPTLINK and describes in detail the effects of each.

All options may appear in the Lahey OPTLINK environment variable, on the command line, or in the first line of the indirect file. They are identified by having "/" as the first character. The options all have short names. While the full name of each is listed in this section, only enough characters to uniquely identify the option need be supplied. Option names may appear in either upper or lower case.

When a Lahey OPTLINK action may be controlled by both a configuration setting and an option command, the option command always overrides the configuration setting. Option commands provided on the command line or in an indirect file always override those read from the Lahey OPTLINK environment variable.

Several of the options require numeric values to be supplied; these are indicated in this section by the suffix ":NNN" in the heading. In all cases the numeric value must be separated from the option name by the colon. The numeric base is, by default, decimal. If the value begins with "0", its base is taken to be octal, and if it begins with "0x" or "0X", hexadecimal is assumed. That is, the standard C-language base convention is assumed by Lahey OPTLINK.

F.5.1 Checksum Options

These three options control how Lahey OPTLINK deals with checksum calculations for both object modules and the generated ".EXE" file. Compatibility with all other linkers is not possible with one setting, since the other linkers are inconsistent in their treatment of this capability.

<u>**Checksum Control**</u> Only Lahey OPTLINK permits you to tailor the linker's checksum actions to match your normal operating conditions.

/CHECKA[BORT]
/NOCHECKA[BORT]

This switch controls warning versus abort on object-module checksum errors. The default setting is "NO". Neither TLINK nor MS-LINK even check the checksum.

/CHECKE[XE]
/NOCHECKE[XE]

This switch controls generation of the ".EXE" checksum word. The default setting is "NO". TLINK doesn't, MS-LINK does.

/CHECKS[UM]
/NOCHECKS[UM]

This switch controls checking of the object-module checksum byte. The default setting is "NO". MS products often generate invalid checksums.

F.5.2 Filesize Options

These options control the size of the output file Lahey OPTLINK writes, and of the final program executing in RAM.

/CO[DEVIEW]
/NOCO[DEVIEW]

This switch controls appending CodeView information to the end of an ".EXE" file. The default setting is "NO".

NOTE: Lahey FORTRAN-generated object files do not support CodeView.

/CP[ARMAXALLOC]:NNN

This switch sets the maximum number of bytes that the linked program will occupy in RAM when loaded. It requires the colon, followed by the number of 16-byte paragraphs (in the range 1 to 65535) to permit the program to occupy. If the program requires more space, Lahey OPTLINK will (like other linkers) set the allowable RAM usage to the actual amount required by the program. If this switch is not used, Lahey OPTLINK will set the RAM requirement to its maximum possible value, causing the program to use all available RAM at run time. The numeric base is, by default, decimal. If the value begins with "0", its base is taken to be octal, and if it begins with "0x" or "0X", hexadecimal is assumed.

/ST[ACK]:NNN

This switch sets stack size. If used, it also actually defines a stack segment (in DGROUP if /DOSSEG is also set to "YES") if one wasn't specified in the ".OBJ" file. The number, which may not exceed decimal 65535, is required, and specifies the size in bytes for the stack. If the value begins with "0", its base is taken to be octal, and if it begins with "0x" or "0X", hexadecimal is assumed.

F.5.3 Language Compatibility Options

/DE[FAULTLIBRARYSEARCH]
/NODE[FAULTLIBRARYSEARCH]

This switch controls how Lahey OPTLINK handles comment records in an ".OBJ" file requesting a library search. The default setting is "YES".

/IG[NORECASE]
/NOI[GNORECASE]

This switch controls case significance for publics and externs. The default setting is "YES" which causes case to be ignored.

/DO[SSEG]
/NODO[SSEG]

This switch controls segment sequence. If "YES", the sequence of segments in the linked ".EXE" file will be (first) all segments with a class name ending in "CODE", (second) all segments NOT included in DGROUP, and (last) the segments of DGROUP, which will in turn be sequenced as BEGDATA, all except BEGDATA BSS and STACK, BSS, and finally STACK. If "NO", segments appear in the sequence in which they are encountered while linking. The default setting is "NO".

F.5.4 Operational Options

/EC[HOINDIRECT]
/NOEC[HOINDIRECT]

This switch controls echoing of indirect-command file input lines to the console. The default setting is "NO".

/PAU[SE]
/NOPAU[SE]

This switch allows swapping of diskettes just before Lahey OPTLINK writes binary output data. The default setting is "NO".

/HE[LP]

This switch displays a list of parsed switches.

F.5.5 Map File Options

/L[INENUMBERS]
/NOL[INENUMBERS]

This switch controls addition of line-number information to the MAP file. The default setting is "NO".

/M[AP]
/NOM[AP]

This switch controls addition of public symbols to the MAP file. The default setting is "NO".

/X[REF]
/NOX[REF]

This switch controls addition of cross-reference information to the MAP file. The default setting is "NO".

F.5.6 Switches Not Implemented

These switches (recognized by the Microsoft Linker) are parsed, but ignored:

/PACKCODE
/FARCALLTRANSLATION
/QUICKLIB
/HIGH
/DSALLOCATE
/EXEPACK
/BATCH
/INFORMATION
/NOEXTDICTIONARY
/OVERLAYINTERRUPT
/SEGMENTS

F.6 Linker Error and Warning Messages

This section lists all Error and Warning messages generated by Lahey OPTLINK and describes probable causes and remedies for each. Each message, together with added information such as the symbol name or file name involved, is displayed to the stdout device when its triggering condition is detected. If a log of error logs is desired, DOS redirection may be used to create one. For each error message, the following rules apply:

- if the filename is known, it is printed;
- if the module name is known, it is printed also (useful especially if the filename is ".LIB");
- if the error occurs at a known offset in the file, the offset (in HEX) is printed;
- if a specific ".OBJ" record is being worked on, its record type (a number in HEX corresponding to the standard Intel OMF or some extension) is printed; and
- in every case, the error message number is printed, followed by the message text.

F.6.1 Lahey OPTLINK Warning Messages

Lahey OPTLINK provides several warning messages, which alert the operator to inconsistencies but do not indicate fatal problems.

File Not Found

This warning message is displayed when commands are given to perform any operation with a ".LIB" file which cannot be found. The name of the missing ".LIB" file is appended to the message. Note that the same message, when referring to an ".OBJ" file, indicates a fatal error.

Multiple CODE Segments In Single Module

Codeview information currently does not support multiple code segments in a module.

NOTE: Lahey FORTRAN-generated object files do not support CodeView.

No Stack

This warning message is displayed when no stack segment is supplied. If a ".COM" or ".SYS" file is being generated, it is normal; in other cases, be careful.

Start Previously Specified In Module

This warning message is displayed when more than one start address is specified for a program. Start addresses are specified by putting a label after the END statement, like:

```
END    MY_ROUTINE
```

Only one module should specify a start address. If more than one is found, Lahey OPTLINK uses the first one encountered.

Too Many Symbols for MAP File

This warning message is displayed when more public symbols are supplied than Lahey OPTLINK is able to sort. The map file is generated unsorted.

F.6.2 Lahey OPTLINK Error Messages

Errors which prevent successful completion of a Lahey OPTLINK run are described in the following subsection. Processing stops immediately upon detection of any of these fatal errors.

Bad Checksum

This message appears when Lahey OPTLINK detects a checksum error in any module, if checksum verification has been enabled. The module is probably corrupted, or it was generated by MASM.

OPTLINK Feature	The ability to select responses to a bad checksum is unique to Lahey OPTLINK.

Below 100H Cannot Be Initialized

This fatal error message is displayed when the COM-file option is selected and an attempt to initialize data areas at an address lower than 0100h is detected.

BYTE Out of Range

This fatal error message is displayed when code such as "mov al,ext_abs" where ext_abs is greater than 255, is encountered.

Cannot Create File

This fatal error message is displayed when any requested output file cannot be created. Most probable causes are that the requested filename contains an invalid character, or that the output disk directory is full (applicable only to root directories).

Cannot Reach ASEG from RELOC or RELOC from ASEG

This fatal error message is displayed when an attempt is made to generate an offset from a relocatable segment to an absolute variable or constant.

Cannot Reach TARGET from FRAME

This fatal error message is displayed when a traditional "fixup overflow" is detected by Lahey OPTLINK. The problem is usually caused by memory-model conflicts, a segment or group being too large, segment order or combining errors (look at the ".MAP" output to diagnose this), incorrect ASSUME statements, or just plain trying to access a variable when no segment register points to it, or trying to do a NEAR call/jmp to a FAR routine. Assemble or compile with linenumbers ON to determine the exact location of the error.

OPTLINK Feature Lahey OPTLINK is unique in its ability to identify the source line number where such a fixup error occurs. This can be of great assistance in running down problems of this type, which are among the most troublesome ones encountered with large systems.

Colon Expected

This fatal error message is displayed when a Lahey OPTLINK option that requires a number is requested, and the ":" or following number is not given. Try the operation again, being sure to supply the colon and all required numbers.

Configured Choice

The message described in this subsection may be treated as either a warning condition, or as a fatal error. The choice is made when you configure Lahey OPTLINK, but may be overridden by option commands.

Data Outside Segment Bounds

This fatal error message is displayed when Lahey OPTLINK finds more data in an object module than the module indicates is there. For example, an object file that indicates 5 bytes of data are to follow, and then provides 6, will cause this error. Most likely cause is that the ".OBJ" file has been corrupted, or that a translator error has occurred.

Disk Full Writing

This fatal error message is displayed when Lahey OPTLINK fails to find enough space on the disk to write the required output file.

DOS Critical Error

This fatal error message is displayed when DOS detects a "critical error" while processing a read or write function for Lahey OPTLINK.

EMS Error

This fatal error message is displayed when Lahey OPTLINK is notified of an EMS error while attempting to access expanded memory. The only known causes for this error are faults in either the EMS driver or the EMS hardware.

File Not Found

This fatal error message is displayed when commands are given to perform any operation with an ".OBJ" file which cannot be found. The name of the missing ".OBJ" file is appended to the message.

FIXUPP Points Past Data Record

This fatal error message is displayed when Lahey OPTLINK detects a situation such as a FIXUPP record instructing that the DWORD beginning at byte 5 of a data record be changed, when the data record contains only 6 bytes (since a DWORD beginning at byte 5 would extend through byte 8). Most likely cause is that the ".OBJ" file has been corrupted, or that a translator error has occurred.

GROUP Cannot Be Both Relocatable and Absolute

This fatal error message is displayed when a segment defined as "SEGMENT AT" is put in a GROUP with a normal relocatable segment.

GROUP Size Exceeds 64k

This fatal error message is displayed when the total size of all segments named within a GROUP exceeds 65,535 bytes.

Illegal Filename

This fatal error message is displayed when Lahey OPTLINK is provided a file name which contains an illegal character.

Illegal Record Syntax

This fatal error message is displayed when Lahey OPTLINK is unable to decipher the syntax of a record in an ".OBJ" file. The most likely cause is that the ".OBJ" file is corrupt, due either to a disk accident or to a translator bug.

Index Range

This fatal error message is displayed when a record in the ".OBJ" file provides an index value which is outside the allowable range of values. The most likely cause is that the ".OBJ" file is corrupt, due either to a disk accident or to a translator bug.

LIB Dictionary too Big

This fatal error message is displayed when a ".LIB" file's dictionary is found to exceed 127K bytes in size. Lahey OPTLINK cannot handle library dictionaries larger than 127K. (We know of no libraries which exceed this limit at the present time.)

Limit of 65535 Base Fixupps Exceeded

This fatal error message is displayed when more than 65,535 segment fixups are encountered in a single executable file. The ".EXE" file format permits only 65,535 segment fixups.

Line Exceeds 500 Bytes Reading

This fatal error message is displayed when more than 500 bytes are read from an indirect file without encountering end-of-line.

LOCATION Not Within FRAME

This fatal error message is displayed when a CALL or JMP is made and the CS ASSUME is not valid.

More Than 1 Stack

This fatal error message is displayed when Lahey OPTLINK detects a second attempt to define a stack segment. Each program can have only one stack segment.

Near Communal Size Exceeds 64k

This fatal error message is displayed when the amount of global data in one small-data-model module exceeds 65,535 bytes.

Not a Valid Library File

This fatal error message is displayed when any file described to Lahey OPTLINK as a ".LIB" file fails to meet the criteria for ".LIB" files.

Nothing Above 100H

This fatal error message is displayed when the COM-file option is selected and no code or data is found at any address equal to or higher than 0100h.

Number Overflow

This fatal error message is displayed when a number greater than 65,535 (16 bits) is supplied on the command line.

OBJ Record Too Long

This fatal error message is displayed when any object record exceeds 1040 bytes in length. The ".OBJ" or ".LIB" file is probably corrupt.

Out of Memory

This fatal error message is displayed when Lahey OPTLINK does not have enough memory to continue operation. Try removing a TSR or two, and link again.

Overflow 32-bit Multiply

This fatal error message is displayed when Lahey OPTLINK's 'far global data' allocation routines encounter an address value which they cannot successfully calculate. While it is documented here for completeness, you are not likely to see this message.

Previous Definition Different

This fatal error message is displayed when any PUBLIC symbol is defined twice, and the second definition differs in any way from the first. The redefined symbol is appended to the message.

PUBDEF Must Have Segment Index

This fatal error message is displayed when Lahey OPTLINK encounters a PUBLIC symbol that has no reference into a defined SEGMENT. The most likely cause is that the ".OBJ" file is corrupt, due either to a disk accident or to a translator bug.

Relocatable Bases Not Allowed In Absolute Mode

This fatal error message is displayed when reference to relocatable data is encountered while processing code in Absolute Mode. That is, the ".OBJ" program(s) cannot reference SEGMENTs or GROUPs, cannot do FAR jumps or calls, and cannot do "DD LABEL", when generating a ".COM" or ".SYS" file, unless the target segment has been declared to be absolute (segment at).

Segment Already In Different GROUP

This fatal error message is displayed when Lahey OPTLINK detects a second attempt to assign any segment to a GROUP. Each segment can belong to only one GROUP.

Segment Not In that GROUP

This fatal error message is displayed when the line-number information stored in the ".OBJ" file is bad or when a translator error is detected.

Segment Size Exceeds 64k

This fatal error message is displayed when the total size of any segment exceeds 65,535 bytes.

Short JMP Out of Range

This fatal error message is displayed when the destination of a Short JMP is more than 125 bytes from the location of the instruction.

Start Address Must Be 100H

This fatal error message is displayed when the COM-file option is selected and a starting address (END START_ADR) other than 0100h is specified.

Symbol Name Exceeds 127 Characters

This fatal error message is displayed when any symbol name exceeds the 127-character limit. The ".OBJ" or ".LIB" file is probably corrupt. The too-long symbol is appended to the message.

Symbol Undefined

This fatal error message is displayed when a symbol remains undefined after all input files, including libraries, have been processed.

Too Many Groups, Segments or Externals in Module

This fatal error message is displayed when Lahey OPTLINK encounters more than 4090 groups + segments, or more than 4090 external symbols, in a single module.

Too Much PUBDEF Data for Debug Info

This fatal error message is displayed if Lahey OPTLINK detects a greater number of PUBLIC symbols defined than Codeview is able to handle, and the Codeview option has been enabled.

NOTE: Lahey FORTRAN-generated object files do not support CodeView.

Unexpected End of File

This fatal error message is displayed when Lahey OPTLINK encounters the end of any input file before finding a MODEND record. The most likely cause is that the ".OBJ" file is corrupt, due either to a disk accident or to a translator bug.

Unknown FIXUPP Frame Type

This fatal error message is displayed when Lahey OPTLINK is unable to determine the frame type of a FIXUPP record in an object module. Most likely cause is that the ".OBJ" file has been corrupted, or that a translator error has occurred.

Unrecognized B2 Record

This fatal error message is displayed when Lahey OPTLINK finds a type B2 record in its input files and is unable to determine the meaning of the record. Most likely cause is that the translator has added an undocumented extension to the B2 record usage.

Unrecognized Communal Syntax

This fatal error message is displayed if Lahey OPTLINK detects any error while processing a type B0 (Communal Data) record.

Unrecognized FIXUPP Type

This fatal error message is displayed when the reported situation is detected. Most likely cause is that the ".OBJ" file has been corrupted, or that a translator error has occurred.

Unrecognized Record

This fatal error message is displayed when Lahey OPTLINK cannot recognize the type of a record in an ".OBJ" file.

User ABORT

This fatal error message is displayed when the user aborts operation by pressing Control-C or CTRL-BREAK.

RESERVED FOR YOUR NOTES

SOURCE ON-LINE DEBUGGER G

LAHEY P77L PROFILER H

Not All Lahey Language Systems Include the Profiler.

SOLD DEBUGGER &
P77L PROFILER

SOURCE ON-LINE DEBUGGER G

LAHEY P77L PROFILER H

Not All Lahey Language Systems Include the Profiler.

G SOURCE ON-LINE DEBUGGER

The Source On-Line Debugger ("SOLD.EXE") is a debugging tool which allows a user to interact with the execution of an F77L program at the FORTRAN source level. Using SOLD enables a programmer to trace the execution, stop the execution at any statement, view the FORTRAN source file and display and modify the values of variables and arrays in the current program unit.

Recompilation and relinking are not required for a program to go from debugging mode to production mode, or vice versa. SOLD executes the production program. The extra information required by SOLD is put into a file with the same filename as the source file and the extension ".SLD". When debugging is complete, the ".SLD" files can be deleted.

G.1 Overview of SOLD

To use SOLD, type:

SOLD [@<comfile>] <pname> [<cl>]

Where:

"comfile" is the filename containing SOLD commands;

"pname" is the name of the ".EXE" file for the program to be run; and

"cl" is the optional command line text that is passed to the executing program.

SOLD then displays the version number of the compiler that compiled the main program. The prompt ">>" is displayed whenever SOLD is ready to accept commands.

You can use the SOLD command file feature to set up a debugging session. The default extension for "comfile", the SOLD command file, is ".CMD". If your command file has no extension, terminate the command filename with a period. For example:

SOLD @sldcom. mysource

If SOLD requires more input after reading the command file, "sldcom", the normal prompt ">>" is displayed.

In order to take full advantage of the features of SOLD, the program units being debugged must still have their ".SLD" files available. Even if the ".SLD" file is not available, SOLD can still set breakpoints at the statement level and trace execution by line number.

Use the **Ctrl-S** key combination to temporarily halt the display of SOLD output. Use **Ctrl-C** to stop a display, or if your program is running, to cause a break at the next line SOLD can control (following the next input/output statement).

If an error is detected by F77L during execution of a program being run under SOLD, a break to SOLD occurs so that you can examine the current values of variables. Program execution may not be continued after the error.

Note that SOLD I/O may conflict with program graphics.

The available SOLD commands are: **Active, Break, Find, Help, Input, List, Mode, Next, Print, Quit, Redirect, Store, Trace, Version, eXecute** and **Restart (&)**.

G.1.1 SOLD Command Syntax and Conventions

Uppercase and lowercase letters are considered to be equivalent, except when entering CHARACTER values in an INPUT command.

SOLD command codes are one or two characters long.

Blanks are optional between command codes and arguments.

An asterisk specifies all program units.

A minus sign (–) preceding a break or trace command deletes any matching command from the SOLD tables.

Blanks and commas separate the arguments to commands that use more than one argument.

Several commands of the same type may be entered without repeating the command code by separating the command argument sets by commas.

Several commands of various types may be entered on the same line by separating them by semicolons or slashes.

A slash after or between commands causes program execution to begin. Any commands after the slash are not examined by SOLD until after the next breakpoint is encountered.

A line number range is indicated in the command syntax as <range>. A range is specified by either a single number or the starting number followed by a colon, followed by the ending number. The syntax is:

```
from[:to]
```

The line numbers are the actual numbers of the lines in the source file where the statements begin.

The line number of the next statement to be executed when execution is halted is called the <u>current line</u>. The program unit containing that line is called the <u>current program unit</u>. The source file containing that program unit is called the <u>current source file</u>. When SOLD first requests input, the current program unit is the main program, and the current line is the first executable line in the main program.

G.2 SOLD Command Descriptions

G.2.1 Active Command

Syntax

A

This command shows all of the program units that are currently active, including a traceback to show where they were invoked. An "active" program unit is one that has been invoked more times than it has been exited. The traceback is similar to that which is displayed when an error is detected during execution of an F77L program.

Example:

A A traceback of currently active subprogram invocations is displayed.

G.2.2 Break/Trace Commands

A Trace command causes a line to be displayed when a designated condition is satisfied. A Break command is essentially like a Trace command except that break causes execution to stop after the trace information is displayed. Breaks or Traces can be done on entry to or exit from a subprogram or on a range of line numbers. Once a Break or Trace command has been given, the command continues to be in effect until it is deleted. Enter a minus sign (–) before the command to deactivate the command.

If you do not specify a program unit in the command, the current program unit is assumed.

Entering a Break or Trace command without any arguments causes SOLD to display the active Break and Trace commands.

Break/Trace Entry

Syntax

{B|T}E [{<punit>|*} [%<n>]]

Where:

<punit> is the name of a program unit; and

<n> is the number of a stored command string (see Store command below).

The Break/Trace Entry command allows you to follow the execution of a program based upon when subprograms are entered. This is useful for getting an overview of the program flow without having to get involved in the individual statements.

Examples:

BE JOE,SUB1	Break on entry to "JOE" or "SUB1".
TE *	Trace the entry to all subprograms.

Break/Trace First Executable Line

Syntax

{B|T}F [{<punit>|*} [%<n>]]

Where:

<punit> is the name of a program unit; and

<n> is the number of a stored command string.

This command allows you to stop or trace on the first executable statement in a program unit. While not quite as fast as {B|T}E, interrupting at this point allows access to all adjustable arrays and other dummy arguments.

Examples:

BF thissub	Stop at the first executable statement of "thissub".
TF *	Trace the first executable statement of all subprograms.

Break/Trace Exit

Syntax

{B|T}X [{<punit>|*}   [%<n>]]

Where:

<punit> is the name of a subprogram; and

<n> is the number of a stored command string.

This command allows you to follow the execution of a program based upon when subprograms are exited.

Example:

TX JOE	Trace when exiting "JOE".

Break/Trace Line

Syntax

{B|T}L [<range>] [<punit>|*] [%<n>]

Where:

<range> is a line number range;

<punit> is the name of a program unit; and

<n> is the number of a stored command string.

This command allows tracing the execution of a group of program lines or breaking before executing any specified lines. Any range of lines in the current program unit or any other program unit may be specified.

Examples:

BL 110	Break at line 110 in the current program unit.
TL SUB2	Trace all lines in "SUB2".
TL *	Trace all lines.
BL 347 JOE	Break at line 347 in "JOE".
BL SUB1,240 *	Break at any line in "SUB1" or at line 240 in any program unit.
BL 10 SUB1 %2	Assume stored command string "2" is "I X=3.5;P Z". Then stop each time line 10 of program unit "SUB1" is executed; assign 3.5 to X, print the value of Z, then prompt for more input.

Break/Trace Change of Value

Syntax

{B|T}C <item> [<range>][<punit>|*] [%<n>]

Where:

<item> is a variable or array element known to the current program unit;

<range> is a line number range;

<punit> is the name of a program unit; and

<n> is the number of a stored command string.

This command aids in finding out where a variable or array element changes in value, and what values it changes to. The checking for change of value can be limited to any range of lines in the current program unit or any other program unit. The item whose value is being checked must have been specified in the current program unit, or in a program unit in the active calling chain.

SOLD attempts to indicate the line of code that caused the change and indicates the line with a message of the form:

```
I <changed from> 0 <to> 1 <in> _MAIN <at line> 5:    I = 1
```

If SOLD doesn't know the exact line that caused the change, the message has the form:

```
I <changed from> 0 <to> 1 <discovered> <in>_MAIN <at line> 2:
CONTINUE
```

The typical reason for SOLD's not knowing the exact line is that the variable was changed by a CALL or an external function invocation in the line immediately before the indicated line.

Examples:

BC A *	Halt execution (break) if the value of the variable "A" changes, and display the old and new values.
TC B(3) SUB2	Whenever the array element B(3) changes during execution of the subprogram "SUB2", display the line number and the old and new values.
TC L 100:140	Whenever the variable "L" changes during execution of lines 100 through 140 of the current program unit, display the line number and the old and new values.

BC A, B(4)	Halt execution (break) if the value of either the variable "A" or the array element "B(4)" changes in the current program unit, and display the old and new values.
TC count * %2; TX sub2	Trace whenever "count" changes in any program unit, then execute command string "2"; also trace whenever exiting subroutine "sub2".
TC I %3	If stored command string "3" is "P X(I),Y(I)", then whenever "I" changes value in the current program unit, display X(I) and Y(I).

Break/Trace When a Condition is True

Syntax

{B|T}W [.NOT.]<item> [<cond><value>] [<range>] [<punit>|*] [%<n>]

Where:

[.NOT.] is a FORTRAN operator;

<item> is a variable or array element known to the current program unit;

<cond> is a conditional operator;

<value> is the value the variable is compared to;

<range> is a line number range;

<punit> is the name of a program unit; and

<n> is the number of a stored command string.

The conditional operator <cond> is one of the following:

.EQ.	=
.NE.	< >
.GT.	>
.GE.	> =
.LT.	<
.LE.	< =

For LOGICAL variables, <cond> and <value> are not required. COMPLEX and LOGICAL variables may be compared only for equal and not equal.

This command allows tracing or stopping execution when a particular condition is true. As with other similar commands, the checking can be limited to a range of lines and/or program units. If a range is not specified, all lines of the current program unit are assumed. The item being checked must have been specified in the current program unit.

If a LOGICAL variable is specified without a condition and value, the condition is simply the value of the variable. If the operator ".NOT." is specified, then the opposite of the specified condition is tested for.

When BW is used, a break occurs as soon as the condition becomes true. Another break will occur only if the condition becomes false, then true again. If you are only interested in the first time the condition becomes true, it may be advisable to delete the BW command to avoid slowing SOLD down unnecessarily.

Examples:

BW i = 5	Halt execution when loop counter "i" is 5.
BW a(3)>7.5e4	Halt execution when the third element of array "a" becomes greater than 75000.
BW .NOT. FLAG	Halt execution when LOGICAL variable "FLAG" is false.

TW i.LT.99 BB	Trace all lines in subprogram "BB" whenever "i" is less than 99.

BW count .GT. 5 * %5; BL sub1 30:35/

	Stop whenever "count" is greater than 5 in any program unit, then execute command string "5"; also stop at lines 30 through 35 in subroutine "sub1".

<u>Break on Stop</u>

<u>Syntax</u>

BS

This command will cause SOLD to break at program termination. The program unit, where the STOP statement was executed, will be the current program unit for listing and dumping variables. The current line will always be the END statement regardless of where the STOP statement is in the program unit.

<u>Displaying Break/Trace Tables</u>

<u>Syntax</u>

{B|T} [E|F|X|L|C|W]

The Break and Trace commands currently in force can be displayed by entering the Break or Trace command with no arguments.

Examples:

BE	Display all Break On Entry commands.
TL	Display all Trace Line commands.
B	Display all Break commands.

Deleting Break/Trace Commands

Syntax

−{B|T} [E|F|X|L|C|W]

A minus sign (−) preceding a command negates that command, i.e., deletes it from the list of active commands. Any BW or TW command may be deleted by entering "−BW" or "−TW", respectively. To further qualify which commands to delete, a variable or array element name optionally followed by a program unit name and/or a line number range may be specified after the −BW or −TW. Conditions and values may not be specified when deleting commands. Examples:

−BE JOE	Don't break on entry to "JOE" anymore.
−T	Delete all current Trace commands.
−BL SUB1	Delete all Break On Line commands in "SUB1".
−TL 20:30 SUB2	Delete all line traces in "SUB2" from line 20 through line 30.
−BW a(3)	Delete all BW commands for "a(3)".
−BC INDEX	Don't break when "INDEX" changes anymore.
−TW A _MAIN	Delete all "Trace When" commands active only in the main program.

G.2.3 Find Command

Syntax

F ['<string>' [<range>] [<filename>]]

Where:

<string> is the line of text to be found;

<range> is the range of lines to search for <string>; and

<filename> is the source file to search.

This command instructs SOLD to search for a line of text in a source file. You may optionally specify a range of lines to search and a source file other than the current source file to be searched.

Specify **F** to continue searching with the previously specified Find command.

Specify a carriage return (**ENTER**) as the first or last character in the string to indicate checking only at the beginning or end of the line, respectively.

Examples:

F '100' 300:400	Find the first occurrence of the text "100" in lines 300 through 400 in the current source file.

NOTE: Find treats uppercase letters as being equivalent to lowercase letters.

G.2.4 Help Command

<u>Syntax</u>

H [<cletter>|<hpage>]

Where:

<cletter> is the first letter of a SOLD command; and

<hpage> is the page number of a SOLD Help Screen.

Use **H<cletter>** to see the Help Screen for the indicated SOLD command. Use **H<hpage>** to see a specific page of the SOLD help information.

Entered as **H**, this command displays a summary of the syntax and available commands in SOLD. Press **H** repeatedly to see all of the pages of help information.

Example:

H	Display a summary of SOLD commands.
HB	Display help on setting breakpoints.
H3	Display screen 3 of SOLD help information.

G.2.5 Input Command

<u>Syntax</u>

I <item> [(<subscript>)] [(<sstring>)] [=]<value>

Where:

<item> is a variable or array element known to the current program unit;

<subscript> is an optional subscript set;

<sstring> is a substring specifier; and

<value> is the value to input into <item>.

This command allows you to input a constant value to a variable, substring or array element. Complex values require the use of the equal sign. The variable or array must have been referenced in the current program unit or in another program unit that is currently active (see Active command above). The subscript and substring specifiers may be constants or INTEGER variables.

Sometimes SOLD is not able to modify a variable because it is being carried in a register. It will appear to have been modified, but when you execute the program, your assigned value may be overwritten. A variable can always be modified at a label or at the beginning or end of a program unit.

Examples:

I A "Help"	Set the CHARACTER variable "A" to the value "Help".
I B(4) 33.5e2	Set the numeric array element "B(4)" to the REAL number 3350. Type conversions will be performed by SOLD, if necessary.
I C=(10,3.5)	Set the COMPLEX variable "C" to 10 + 3.5i. In this example the "=" sign is not optional.
I L(J).true.	Set the LOGICAL array element "L(J)", where "J" is a non-COMPLEX numeric variable in the current program unit, to the value ".true.".
I char(2)(7:i) = "new"	Set characters 7 through "i" in the second element of "char" to "new".

G.2.6 List Command

Syntax

L [<range>|{[<range>]<filename>}]

Where:

<range> is the range of line numbers to list; and

<filename> is the name of a source file.

You can list lines from the current program unit source file or from any other ASCII file. If you enter **L** without arguments, all lines of the current program unit are listed, one screen of text at a time. If output is redirected to a printer, the whole program unit is listed. Use the "range" argument to specify a portion of the file to be listed. Use the argument "file" to list an ASCII file other than the current program unit source file.

Specifying the at sign character "@" as the only argument will cause the current line to be displayed. When listing the current program unit (unless SOLD output is redirected), only a screenful of lines will be displayed for each **L** command. Two or more consecutive **L** commands will display two or more screenfuls of lines in the current program unit. When the lines are displayed, an "@" will follow the line number of the current line.

Examples:

L	List all lines of the current program unit, one screenful at a time.
L 101:140	List lines "101" through "140" in the current source file.
L INCL.FOR	List all lines of the file "INCL.FOR".
L 10:15 S.FOR	List lines "10" through "15" of the file "S.FOR".

G.2.7 Mode Command

Syntax

M [V|B|L|A|N|H|C]

This command controls how much information is displayed and the format of the display when a trace or break is executed.

The line display modes are the following:

V	Very brief; just display the program unit name.
B	Brief; display the line number and the program unit name.
L	Line; display the first line of each statement. This is the default line display mode.
A	All; display all lines up to the next statement.

The print display modes are the following:

N	Normal; display variables according to their types. This is the default display mode.
H	Hexadecimal; display all variables as hexadecimal bytes.
C	Character; display all variables as ASCII characters.

If you specify **M** without arguments, the line and print display modes are displayed.

Examples:

MV	Display the program unit name only.
ML	Display the line number and first line of the statement.
MH	Display all variables as hexadecimal values.
M	Display the current line and print display modes.
MA	Display the line number and all lines of the statement. Any nonexecutable statements immediately following will also be displayed.

G.2.8 Next Command

Syntax

N

The Next command advances to the next executable line of code. Next functions as a single-step.

Example:

N	Advance to the next executable line of code.

G.2.9 Print Command

Syntax

P <item> [(<subscripts>)] [(<sstring>)]

Where:

<item> is a variable, array or array element known to the current program unit;

<subscripts> is either one subscript or a range of subscripts in the form <from:to>; and

<sstring> is a substring specifier.

This command causes SOLD to print the value of a variable, substring, array or partial array. The variable or array must have been referenced in the current program unit or in another program unit that is currently "active" (see Active command, above).

A subscript or substring specifier used in the Print command may be a constant or an INTEGER variable.

If an array subscript is outside the bounds of the array, or the number of subscripts is not equal to the number of dimensions of the array, the message:

Invalid Array Operation

is displayed. Adjustable-dimensioned arrays, character dummies and some other dummies in program units containing ENTRY statements cannot be referenced at the entry point to a subprogram, because the pointers for them have not been set up yet. They will be available at the first executable statement.

Examples:

P A	Print the value of the variable "A".
P B	Print the values of each element of the array "B".
P B(J)	Print the value of the array element "B(J)", where "J" is a non-COMPLEX numeric variable in the current program unit.
P matrix(3:4, 1:3)	Print elements (3,1), (4,1), (3,2), (4,2), (3,3) and (4,3) of "matrix".

G.2.10 Quit Command

Syntax

Q

This command causes SOLD to exit to DOS.

Example:

Q	Quit this SOLD debugging session.

G.2.11 Redirect Command

Syntax

R [<filename>][,[<filename>]]

Where:

<filename> is the name of a file to receive the output.

This command allows redirection of the output from SOLD to a file. Output may be directed to two files or devices. If the first <filename> is not specified, SOLD output is directed to the console. If the comma is specified but a second <filename> is not specified, output only goes to the first file.

Examples:

R LPT1	Send SOLD output to the "LPT1" printer.
R TEST.OUT	Send SOLD output to a file named "TEST.OUT".
R	SOLD output returns to the console.
R, TEST.LOG	Send SOLD output to the console and to file "TEST.LOG".
R,	SOLD output goes to the console only and the second output file is cancelled.
R TEST.OUT,LPT1	SOLD output goes to the file "TEST.OUT" and the "LPT1" printer.

NOTE: The "&" Restart command cancels all redirection.

G.2.12 Store Command

Syntax

S <n> <text>

Where:

<n> is the number of a command string; <n> must be in the range of 0–19; and

<text> is the text of a command line.

The Store command causes a line of command text to be stored in a SOLD table for later use. Use the argument <n> to identify the command's place in the table. All text following <n> to the end of the Store command line is stored as the command text (see the eXecute, Trace and Break commands).

Example:

```
S 10 P I,J,K,X(3:5,I);MH;P J;MN
```

Store the following sequence of commands as command string "10":

```
P I,J,K,X(3:5,I)
MH
P J
MN
```

G.2.13 Version Check Command

Syntax

V [Y|N]

The Version Check command tells SOLD whether or not to check that all program units have been compiled with the same version of F77L. This could be important if the versions were not totally compatible or if a bug was corrected in a later version.

Examples:

VY	YES, report if a subprogram and the main program unit were compiled by different versions of the compiler. "Y" is the default.
VN	NO, don't report different versions.
V	Display "Y" or "N" for selected version checking.

G.2.14 eXecute Command

Syntax

X<n>

Where:

<n> is the number of a stored command string.

This command allows you to execute a stored command line (see Store command, above).

Example:

X5	Execute command string "5".

G.2.15 Restart Command

Syntax

&

This command will restart the program being debugged at the beginning of the main program. All variables will be initialized to their startup values. Note that Redirection will be cancelled by this command.

Example:

&	Restart at main program.

G.3 Debugging with SOLD

This section explains techniques for debugging with SOLD and contains examples of how to use SOLD in actual debugging situations, which may provide some insights into efficient ways to find program errors.

A command file can be used if the same commands need to be entered several times. An entire debug session can be put into a command file and then be modified as needed, or the command file can be used to set up the debug session by using the Store command, setting breakpoints, and so on.

If the command file has fewer "/" commands than the number of breakpoints reached during execution of the program, then SOLD issues the ">>" prompt to request more commands.

The examples below are followed by explanations of the effect of the SOLD command line.

Examples:

```
BE SUB1/P A(I)/
```

Set a break on entry for subprogram "SUB1", then start executing. When "SUB1" is entered, print the current value of the array element "A(I)", then continue executing until "SUB1" is entered again or the program is done.

```
BX SUB2///TL *;BE SUB2/
```

After exiting subprogram "SUB2" three times, start tracing all statement lines until "SUB2" is entered again; then halt execution.

In debugging large programs, it is often important to have a strategy for pinpointing the location of the error as quickly as possible. The following scenarios illustrate strategies for using SOLD to find program errors quickly.

Scenario 1:

In a large program, the variable "J" is set to "1" at line 23 of the main program, but has the unlikely value "262399" at line 135. Also, because of the size and complexity of the program, just tracing all of the changes for variable "J" would take a long time. First we find out which line in the main program causes the strange value with the command line:

TC J 23:135;BL135/

From this we find that "J" is set to some reasonable values, then becomes "262399" at line 122, after the statement "CALL SUB1(J, ...)". Then we find where in SUB1 "J" is set to "262399" by quitting and restarting SOLD and entering the command line:

BL 121/TC J SUB1;BL 122/

From this we find that "J" is set to "262399" in SUB2 at line 453. After a couple of iterations of this method, we find "J" gets the bad value in the statement "X(44) = 3.5" in "SUB5". This makes us suspect that "X" was dimensioned incorrectly. By checking the source code we find that the "X" is a dummy argument dimensioned as (50) in "SUB5", but was originally defined in the main program with a dimension of (40). Dimensioning "X" to (50) in the main program eliminates the error.

Scenario 2:

The variable "Z" has an unreasonable value in it after the statement "Z = A + B – (C/D)", which is at line 135 in subroutine "SUB3". First we use the following SOLD command line:

BL 135 SUB3/P A,B,C,D;Q

from which we find that the value in "B" looks incorrect. The "Q" causes SOLD to terminate after printing the values of the variables. We can then find how "B" gets set by rerunning the program under SOLD and entering the following command line:

```
R PRN;BE SUB3/TC B *;BL 135/Q
```

which redirects the SOLD output to the printer (PRN), goes to the beginning of subroutine "SUB3", then traces how the variable "B" changes. If the source of the problem is not yet obvious, this technique can be used to find why other variables have incorrect values in them, until the bug is found.

G.4 Using SOLD With Subdirectories

F77L creates the SOLD information file ".SLD" in the same directory as the source file unless specified otherwise on the command line. If you want to run SOLD on an executable file in a different directory than the source and ".SLD", just specify the entire pathname when compiling the source. SOLD will then be able to find the files it needs regardless of the directory it is run in.

Examples:

To compile, enter:

```
F77L   C:\PROJECT\SOURCE\MAIN,C:\PROJECT\OBJ\MAIN
CD \PROJECT\OBJ
OPTLINK  MAIN;
```

After Linking, enter:

```
SOLD  MAIN
```

H LAHEY P77L PROFILER

Lahey Computer Systems' P77L Profiler is a powerful tool that allows you to analyze your program's performance and helps you identify where to optimize its efficiency. Using the Profiler for post-development analysis can help guarantee that all of a program's statements are executed as well as help increase program execution speeds.

A study of FORTRAN programs conducted by Donald Knuth of Stanford University found that a well-done optimization can increase program speed four to five times. The study also found that less than 4% of the code in the programs sampled accounted for more than half of the execution times.

Optimizing a program may be a more cost-effective way to increase its execution speed than upgrading to a faster PC. For example, it is possible that a well-optimized program may execute faster on an 80286-based PC than a non-optimized version of the same program running on an 80386-based PC.

The Profiler aids debugging by enabling programmers to count program unit and FORTRAN statement invocations. The Profiler's Time and Count Functions can help identify where inefficiencies exist in source code. For example, a statement block with a high execution time may suggest where you should concentrate your efforts to achieve the most dramatic execution speed improvements. Statements with high counts may also suggest where to improve code. Statements with zero counts may reflect unused or untested sections of code which possibly could be deleted.

Sometimes simple changes in a program can cause substantial improvements in its efficiency. By making a change and observing the Profiler results, a programmer can see the consequences of the change.

H.1 Profiler Function Overview

The Lahey Profiler program analysis functions are:

Active	**Help**	**Quit**
Break	**List**	**Screen**
Count	**Mode**	**Version**
Dump	**Post Dump**	**Time**
Exclude Unit		

Many of these functions are identical to F77L's SOLD Debugger. If you are already familiar with SOLD, you know how to use many Profiler functions.

Most functions, except Active and Quit, have Command Types and/or Command Arguments allowing you to specify exactly what you want each command to do. To invoke the Profiler, enter:

P77L [FILENAME]

Once the Profiler's "P77L>>" prompt appears, enter the functions you want performed as explained in the **Functions Directory** section H.5.

The **Functions Directory** section discusses:

Command Description,
Command Types,
Command Arguments,
Command Examples,
Command Constraints,
Command Qualifiers, and
Help Commands

A simple program example called "DEMO.FOR" is included on the diskette. The examples in the Functions Directory show how the functions work in terms of their results when applied to the sample program.

H.2 Command Syntax Conventions

This section defines the Profiler Command Syntax Conventions. The commands used by the Profiler are single-letters followed by a set of Command Types and/or Arguments.

Alphanumeric Input

The Lahey Profiler is not case sensitive. All input can contain either upper or lower case characters. There is a 31 character limit to program unit names. Program unit name syntax is identical to F77L.

Asterisks

Asterisks, "*", are wildcards used to specify all units of a given type. For example, break on entry on all program units, type:

```
BE *
```

Blanks

Blanks are not significant in a command line. You may use blanks, however, to distinguish commands using more than one argument.

Commas

Multiple entries can be specified by separating the entries by commas. For example:

```
BE sub1,sub2
```

means Break on Entry in both the "sub1" and "sub2" sub-routines.

Colons

A colon is used to specify the range of an operation. For example:

```
BL 1:10
```

means Break on all executable lines in the range of 1 to 10 for the current program unit

If you specify a set of inclusive ranges for an operation, the Profiler optimizes the ranges to maximize efficiency. For instance:

BL 1:10, 5:15

The Profiler concatenates the two ranges into one range, i.e., 1:15

NOTE: When possible, the Lahey Profiler maximizes the internal table's efficiency when non-executable code is included within a specified range. In the above example, if the first 4 lines of code in a given range are not executable, the Profiler performs the operation on lines 5:15.

Minus Sign

The minus sign (–) is used to delete a command from the Profiler's Tables. Precede Profiler commands with a "–" (minus sign) to delete the command from the Profiler's active commands.

Semi Colons

Commands on a single line may be separated by a semi-colon, ";". For example, the commands:

BE sub1; BX sub1

are equivalent to:

BE sub1
BX sub1

H.3 Profiler Command Input

There are two basic ways to input Profiler functions: Interactive Input and Command File Input.

H.3.1 Interactive Input

Profiler commands may be input interactively in response to the "P77L>>" prompt. Enter commands as you need them after invoking the Profiler. For example, when you invoke the Profiler by typing:

P77L filename

the "P77L>>" prompt appears. At the prompt enter your Profiler command(s) to perform the desired analysis.

H.3.2 Command File Input

If you use the same sets of commands frequently, create a command file to save time. You then can invoke those commands with one operation. Command files are especially useful for running a specific script in a repeatable fashion. Although you may specify a different extension, command files by default end with the ".PRF" extension.

Determine which types of information you need to see and the operational sequences you desire and create your command files accordingly. Within the command file the Profiler requires a "/" (slash) to execute the program. If a slash is not found, the Profiler prompts the user for input. Invoke the Profiler using the "COUNT.PRF" Command File by typing:

P77L @count[.prf] filename

Where:

"@" is the command file identifier;

"count.prf" is the command file; and

"filename" is the ".exe" file to which the "count.prf" command file is applied.

H.4 Profiler Output

Profiler output is generated upon program termination by the Dump Command. All output from the Profiler is written to a file with the same file name as the source file with extension, ".P77". This output file is always used. The output file with the ".P77" extension can either be printed or saved at the users' choice. The screen echo may be turned OFF using the Screen Command (see section H.5.11). The examples below show the information format followed by explanations.

Startup Line:

P77L Profiler vX.XX fname.exe fname.prf MM-DD-YY HH:MMa
Compiler Version = X.XX
<in> fname <at line>
1: program fname
P77L>>

Where:

"P77L Profiler vX.XX" identifies the profiler and version number;

"fname.exe" is the executable filename;

"fname.prf" is the command filename;

"MM-DD-YY HH:MMa" is the Date/Time Stamp;

"Compiler Version = X.XX" identifies the compiler version used;

"1: program fname" is the program's first statement; and

"P77L>>" is the Profiler prompt.

When the results are output, the filename is displayed with a Date-Time Stamp showing when the ".SLD" file was created. This Date-Time stamp is displayed whenever the source filename changes during the dump. All output is dumped in descending order within each command type.

H.5 Profiler Functions Directory

This section lists Profiler commands in alphabetical order with detailed explanations of their use. Most Profiler functions have Command Types, Command Arguments, Command Syntax, and Command Constraints. Refer to each command's section for complete information about using the command. Help for specific commands can be entered as "Hx", where "x" is the first letter of a Profiler Command.

H.5.1 Active Command

Command Description:

The Active Command operates identically to the SOLD Active Command by showing all currently "active" program units including a traceback to show where they were invoked. The traceback is similar to what F77L displays when it detects an error during execution.

Command Types:

A displays a traceback of currently active subprogram invocations.

Command Example:

At the "P77L>" prompt type:

A

the Profiler response is:

```
<in>  SUB  <at line>
  55:      subroutine sub( i, *, *, *)
<called>   <in>  DEMO     <at line>
  42:      call sub( ivalue, *40, *50, *60 )
P77L>>
```

This shows that the active invocation is line 55 of the program unit "SUB" which was called from line 42 of DEMO.

Help: To get help for the Active Command type: **HA**

H.5.2 Break Command

Command Description:

The Break Command operates identically to SOLD's Break Command by causing program execution to halt when the defined condition is satisfied. Using the Break Command with no arguments displays the currently-defined Break Commands.

When a Break is detected, the Profiler outputs the time/date of the Break. If no program unit is specified when a line range is part of the input (i.e. BL 1:10), the current program unit is the default unit name. Once a Break Command is entered, it stays in effect until deleted with the –B Command.

Command Types:

B Displays currently-defined Break Commands

BE Break upon Entry

BX Break upon eXit

BL Break at Line(s)

Command Arguments:

The syntax for the Break commands are:

BE <sub>[,sub...]
BX <sub>[,sub...]
BL [<range>][<sub>][,[<range>][<sub>]...]

Where:

"sub" is a subroutine name or function; and

"range" is the line number range

Command Examples:

At the "P77L>" prompt type:

BE sub1	This causes the ".EXE" to halt on entry to the "sub1" subroutine.
BX sub2	This causes the ".EXE" to halt on exit from the "sub2" subroutine.
BL 10 sub1	This causes the ".EXE" to halt at line 10 of the "sub1" subroutine.

Command Qualifiers:

* (asterisk) Specifies program units, except on BL where it means at all lines of all program units.

– (minus sign) Before the Break Command deletes it from the Profiler's Tables. For example, –BE sub1 removes the defined BE Command for "sub1" from the Profiler's Tables. –B removes all defined Break Commands.

Command Constraints:

If a program unit is excluded (see the Exclude Command) and the "*" wildcard was used with a Break Command, the excluded unit will not be processed by the Profiler. Using the "*" wildcard character for program unit specification causes the Profiler to perform a "lookup" upon entering each program unit. Type the explicit program unit name to speed up Profiler operations.

Help: To get help for the Break Command type: **HB**

H.5.3 Count Command

Command Description:

The Count Command causes the Profiler to count each invocation of a defined program unit or lines within a specified range. The Count Command also can be used to report all lines which are never executed. Statements with zero counts reflect untested sections of code which could point to a problem in the program or point to code that could possibly be deleted. Statements with high counts may suggest where to emphasize improvements.

Typing the command without arguments displays all of the currently-defined Count Commands. The current program unit is the default unit name if no other was specified when a line range is part of the input (i.e. CL, CZ). Once a Count Command is entered, it stays in effect until deleted with the –C Command.

Command Types:

- **C** Displays currently-defined Count Commands.
- **CU** Counts program unit invocations.
- **CL** Counts invocations of each line within a specified range of lines in the default (current) or a specified program unit.
- **CZ** Reports all lines within a specified range which were never executed.

Command Arguments:

The syntax for the Count Commands are:

C
CU <sub>[,sub...]
CL [<range>][<sub>][,[<range>][<sub>]...]
CZ [<range>][<sub>][,[<range>][<sub>]...]

Where:

"sub" is a subroutine name or function, and

"range" is the line number range.

Command Examples:

At the "P77L>" prompt type:

CU sub1, sub2	to cause the Profiler to count all invocations of the "sub1" and "sub2" subroutines.
CL 10:11 sub1	to cause the Profiler to count all invocations of lines 10 and 11 in the "sub1" subroutine.

Command Qualifiers:

*	(asterisk) Specifies all lines or program units, except on CL where it means at all lines of all program units.
–	(minus sign) Before the Count Command deletes it from the Profiler's Tables. For example, –CU sub1 removes the defined CU Command for "sub1" from the Profiler's Tables. –C removes all defined Count Commands.

Command Constraints:

If a program unit is excluded (see the Exclude Command) and the "*" wildcard was used with a Count Command, the excluded unit will not be processed by the Profiler.

The Count Unit, Count Zero and Count Line Commands impose heavy performance demands on your system and could take longer to complete than other Profiler functions.

Using the "*" wildcard character for program unit specification causes the Profiler to perform a "lookup" upon entering each program unit. Type the explicit program unit name to speed up Profiler operations.

The Count Line Command, i.e. CL, only counts a "DO Loop" statement once, since F77L-generated code tests the "DO" variable at the end of the "DO" loop structure.

When using the CL Command, the Profiler is unable to document the output in the following case: Where F77L generated code has taken more than one line of source code and optimized them so they both point to a single executable statement. In this situation the following results:

> When outputting in the "Brief Mode" (see the Mode Function), if the line range contained is continuous and the number of invocations of these lines are all the same for that range, the report assumes that line to also be of that value.
>
> When the output is in the "List Mode" the count value is output as "~" to denote "unable to count" (non-executable code, e.g. ELSE or ENDIF Statements).

Help: To get help for the Count Command type: **HC**

H.5.4 Dump Command

Command Description:

The Dump Command causes the current program analysis statistics to be dumped to the console. Use the Dump Command to display intermediate results. Refer to the Post Dump Command to define how the Profiler dumps output for the Time and Count Commands upon exit.

Command Types:

- **D** Dumps all Profiler Time and Count statistics.
- **DD** Dumps and deletes all Time and Count statistics.
- **DN** Dumps all non-zero Time and Count statistics.
- **DZ** Dumps all zero Time and Count statistics.

Command Examples:

Use the Dump Command to see how many times a program unit is invoked at an intermediate location rather than using a post-execution report. To do this, set a breakpoint at the intermediate location and then use the appropriate Dump Command.

Help: To get help for the Dump Command type: **HD**

H.5.5 Exclude Unit Command

Command Description:

The Exclude Unit Command excludes all current Time, Count, and Break Commands that use the "*" for the program unit name from any Profiler processing on the given program unit name. Typing the Command with no arguments specified displays currently-defined Exclude Unit Commands.

Command Types:

The Exclude Unit Command is:

EU Displays the currently defined Exclude Unit Commands.

Command Arguments:

EU [program unit]	Excludes a specified program unit from Profiler wildcard processing.

Command Qualifiers:

– (minus sign) Before the Exclude Unit Command deletes it from the Profiler's Tables.

Help: To get help for the Exclude Command type: **HE**

H.5.6 Help Command

Command Description:

The Help Command gives on-screen help messages. Help for specific commands can be entered as "Hx", where "x" is the first letter of a Profiler Command.

Command Types:

The Help Commands types are:

- **H** Displays generalized help messages
- **HA** Displays Help messages for the Active Command
- **HB** Displays Help messages for the Break Command
- **HC** Displays Help messages for the Count Function
- **HD** Displays Help messages for the Dump Function
- **HE** Displays Help messages for the Exclude Command
- **HL** Displays Help messages for the List Command
- **HM** Displays Help messages for the Mode Command
- **HP** Displays Help messages for the Post Dump Command.
- **HS** Displays Help messages for the Screen Command.
- **HT** Displays Help messages for the Time Command.
- **HV** Displays Help messages for the Version Command.

Command Constraints:

The "P77L.HLP" File must be in the same directory as the "P77L.EXE" File.

H.5.7 List Command

Command Description:

The List Command outputs lines from the current program unit source file or any other ASCII file.

Command Types:

L Lists all lines in the current program unit, one screenful of text at a time.

Command Arguments:

The syntax for the List Command is:

L [<range>] filename[.exe]

Where:

"range" is the line number range, and

"filename" is a source filename.

Command Examples:

At the "P77L>" prompt type:

L@	to list only the current line.
L	to list all lines of the current program unit one screen at a time.
L 1:10	to list lines 1 through 10 in the current source file.
L INCL.FOR	to list all lines of "INCL.FOR".
L 1:10 S.FOR	to list lines 1 through 10 of "S.FOR."

Help: To get help for the List Command type: **HL**

H.5.8 Mode Command

Command Description:

The Mode Command sets the output mode for the Count Line results. When the Profiler "dumps" the results from a CL Command, the number of times the lines are invoked can be displayed in one of two modes: the Brief Mode and the List Mode.

Command Types:

M Displays the current mode.

MB (Brief) Displays the line number (range) beside the number of times invoked.

ML (List) Displays the line number beside the number of times it is invoked and the FORTRAN source. When ML is set, all CL data is dumped to the ".P77" File, and not echoed to the console.

Help: To get help for the Mode Command type: **HM**

H.5.9 Post Dump Command

Command Description:

The Post Dump Command defines how the Profiler dumps output from the Count and Time Commands upon exit. The Post Dump Command does not affect the CZ Command.

Command Types:

P Displays currently-defined Post Dump Commands.

PA Dumps all Time and Count statistics.

PN Dumps all non-zero Time and Count statistics (default).

PZ Dumps all zero Time and Count statistics.

Help: To get help for the Post Dump Command type: **HP**

H.5.10 Quit Command

Command Description:

The Quit Command is used to exit from the Profiler session to DOS. The results of a Profiler session are NOT saved. If you want to save the results of the session, use the Dump Command to send output to the ".P77" File before quitting.

Command Type:

Q to exit the Profiler to DOS.

Help: To get help for the Quit Command type: **HQ**

H.5.11 Screen Command

Command Description:

The Screen Command turns the Profiler's console output echoing ON or OFF. The default Screen echoing mode is ON. When Screen echoing is ON, output is directed to the console and the ".P77" File. When Screen echoing is OFF, output is directed to the ".P77" File and not shown on the console.

Command Types:

S Displays the current Screen mode.

SY (YES) Screen echoing ON.

SN (NO) Screen echoing OFF.

Help: To get help for the Screen Command type: **HS**

H.5.12 Time Command

Command Description:

The Time Command causes the Profiler to record an elapsed program execution time for program units or specified line ranges. Statement blocks which have high execution times may suggest where to make improvements.

The Profiler's timing resolution is approximately 1 micro-second. Results can vary due to system overhead. Time Command output is displayed in seconds with exponential notation. A total program execution time is calculated from when the Profiler detects another Time Command request. The total execution time is displayed on the dump of the output in the format "HH:MM:SS.ht."

Typing the command with no arguments specified displays currently defined Time Commands. The current program unit is the default unit name if no other is specified when a line range is part of the input, i.e. TL. Once a Time Command is entered, it stays in effect until deleted with the –T Command.

Command Types:

T Displays the currently active Time Commands.

TU Times the entire program unit.

TL Times specified line ranges within defined program units.

Command Arguments:

The syntax of the Time Commands are:

TU <sub> [,<sub>...]
TL [<range] [<sub>][,[<range>] [<sub>]...]

Where:

"range" is the line number range, and

"sub" is a subroutine name or function.

Command Examples:

At the "P77L>" prompt type:

TL 5:8 sub1	To time lines 5 through 8 of the "sub1" subroutine.
TU sub2	To time the duration of program unit "sub2."

Command Qualifiers:

* (asterisk) specifies all lines or program units, except on TL, where it means at all lines of all program units.

– (minus sign) before the Time Command deletes it from the Profiler's Tables.

Command Constraints:

Due to Profiler overhead, it is recommended that to get the best timing results, you avoid using CL or any Break Line Commands. Results may vary when timing I/O because of DOS I/O overhead. When another application resides in memory, overhead for those applications may also affect timing results.

If a program unit is excluded (see the Exclude Command) and the "*" was used with a Time Command, the excluded unit will not be processed by the Profiler. Using the "*" wildcard character for program unit specification causes the Profiler to perform a "lookup" upon entering each program unit. Type the explicit program unit name to speed up Profiler operations. The results of the Time Command must be considered to be estimates. Timer results can vary each time you run the Profiler on your executable file.

All timing Commands are exclusive to the called routines. When timing a line that calls a subroutine, the called subroutine is not included in the time value of the line at which the call statement appears.

Help: To get help for the Time Command type: **HT**

H.5.13 Version Command

Command Description:

The Version Command turns compiler version checking ON or OFF. The default version checking mode is NO. When version checking is ON, it checks that all program units were compiled with the same version of F77L.

Command Types:

- **V** Displays the currently-defined Version Command.
- **VY** (YES) Reports compiler versions differing from the main program unit.
- **VN** (NO) Suppresses the reporting of differing compiler versions.

Help: To get help for the Version Command type: **HV**

H.5.14 Function Notes

Entry Statement Note

When the program unit name is in reference to a FORTRAN Entry Statement, all processing will be included under the name of the program unit that contains the Entry Statement.

Concatenation Note

When entering a line range at the Break point midway through a program execution, the concatenation process does not save the intermediate results which were already concatenated for the earlier line range. Therefore it is good practice not to define ranges which are to be concatenated during intermediate input to the Profiler.

8086, 8088, 80286 and 80386 Considerations

Due to differences in the machine instruction "fetches", the P77L Profiler may return different timing analysis results with different processors.

H.6 Optimizing Using the Profiler

When optimizing it is helpful to know how the Lahey Compiler generates code for different types of functions and operations. Below are some strategies and examples for isolating program performance information.

H.6.1 Identifying Potential Optimizations

A recommended procedure for easily identifying where to improve performance is as follows:

1. Use the Profiler Time Unit Command, "TU *", to identify the program units requiring the most execution time. This command returns an elapsed time value for all program units. Also, during the time unit process, it is a good idea to count the invocations of all program units using the Count Unit Command, "CU *".

2. After you have determined which program unit(s) consumes the most time, use the Count Line Command, "CL [program unit]", on the unit(s).

3. Use the "TL <range>", Time Lines Command on the most frequently invoked lines.

5. Repeatedly modifying and timing those blocks of code should yield the best optimizations.

H.6.2 Identifying Potential Problems

Using the Profiler's Count Zero Command, "CZ", can isolate potential problems by identifying statements that were never executed. This may reveal problems in the code or may simply indicate obsolete statements which could be deleted, thus making the program smaller and easier to maintain.

Program flow anomalies, i.e. unexpectedly high/low statement counts, may be spotted when examining the output from the Count Line Command listing (see the Mode List Command).

H.6.3 Sample Profiler Session

The example below uses two methods to code formatted character output. It compares the time taken to write out the contents of two character variables. The first variable, "char1", is a single-length character array with 512 elements. The second variable, "char2", is a scalar character of length 512. Both WRITE statements below output these variables. Notice that the time to output "char1" was significantly faster than "char2".

```
C:> p77l chartest
P77L Profiler  v1.20  chartest.exe  5-21-89  3:08p
Compiler Version = 4.00
<break> <in> CHARTEST <at line>
    1:   program chartest

P77L>>  l  1:*
    1:@        program chartest
    2:   c
    3:   c     Timing example of formatted character output
    4:   c
    5:         character*512 char1
    6:         character*1   char2(512)
    7:
    8:         open(8,"scratch",status="scratch")
    9:         write(8,05) char1            !faster
    10:        write(8,10) char2            !slower
    11:        close(8)
    12: 05     format(a512)
    13: 10     format(16(32a1))

P77L>> tl 9,10/
Total program execution time was          0:00:00.61
Fortran source compiled  5-20-87  2:00p  CHARTEST.FOR
  TL  10:10   CHARTEST   .165          seconds 27.1%
  TL  9:9     CHARTEST   .721 e-02     seconds   1.1%
```

Obviously, the method used with CHAR1 was more efficient. Using the Profiler dramatically demonstrates why not to declare the string as an array. An application program that used this type of output would benefit substantially from this particular optimization.

INTERFACES

INTERFACES

I C INTERFACE

There is a high degree of compatibility between F77L and large-model Borland Turbo C, Microsoft C and Lattice C Object Code. Due to on-going product development, F77L to C interface compatibility may vary from version to version of each product. C functions can be called from F77L-compiled code, and F77L subprograms can be called by C functions if the conventions in this appendix are observed. This appendix contains general rules for any interface between F77L and C.

NOTES: Microsoft C main programs may not call F77L-compiled subprograms directly. In order to call F77L subprograms from Microsoft C, a simple F77L main program must be created to call the C main program as if it were a subprogram. Once the C main program has been loaded as a subprogram, it may then call F77L-compiled subprograms if the conventions explained in this appendix are followed.

None of the C language implementations with interfaces in this appendix support the Weitek 1167 or 3167 Chips.

F77L uses its "Fixed-Stack Model" when calling C functions (see section A.5.2).

I.1 Borland Turbo C Interface

Different versions of Turbo C require different "F77LBC.OBJ" and "BCF77L.OBJ" modules. The names of the modules are of the form F77LBC.### and BCF77L.###, where ### corresponds to the Turbo C version number (e.g., 150 for version 1.50). If your version number is not included in a module name, use the highest version number below your version number.

You can either choose to rename the version-specific modules to object files, e.g.

ren *.150 *.obj

or leave the distribution extensions intact, and specify the entire module name at link time.

To compile your Turbo C source use:

tcc –ml –c filename

at the DOS prompt, or the appropriate large-model switch for the environment compiler.

To compile FORTRAN source, use:

F77L filename/NI

plus any other option switches desired.

Link the program using both "F77L.LIB" and the Turbo C Libraries for large data and code. Always specify the F77L Library after all other libraries in your OPTLINK Command. See the examples in the following sections.

In programs where the main module is in C, do not call a C function from an F77L subprogram that was called by a C function.

Suggestion: Whenever possible, test your C functions in a C-only environment and your FORTRAN routines in an F77L-only environment before mixing environments. This should make debugging easier.

In addition to the above restrictions, programmers must be particularly careful when passing arrays between FORTRAN and C. Only one-dimensional arrays can easily be passed to C. For example, a C subprogram called from FORTRAN with an INTEGER array of ten elements could be written:

```
cfunc(int_array)
long int int_array[10];
{
...
```

where the actual size of the passed array will not be checked to verify that it is ten elements long; a second argument is often passed, giving the actual size of the array.

Since FORTRAN stores multidimensional arrays in column-major order, while C stores arrays in row-major order, it is usually not convenient to pass multidimensional arrays to non-FORTRAN routines. If this is done, the element of an n-dimensional array "A" that is declared:

```
A(I1,I2,...,In)
```

in FORTRAN must be accessed with subscripts REVERSED in C, for example:

```
A[In][In–1][...][I1]
```

Also, the first element of a C array dimension is 0, so FORTRAN arrays should be dimensioned (0:n–1) for an array of size n.

NOTES: F77L functions can be called by Turbo C-compiled code if the first C argument is a pointer where the F77L function value is to be stored.

FORTRAN files cannot be directly accessed from other languages, since FORTRAN refers to files using unit numbers, and other languages have a file data type. The general rule is: a given file can be operated on either with FORTRAN code or non-FORTRAN code, but not both. For instance, to do input/output with a C-keyed file from FORTRAN, C routines will have to be called to open the file, add and delete records, etc.

The following structures may be useful for working with F77L data types in C:

```
struct CHARACTER {
        char *text;
        int length;
}

struct COMPLEX {
        float real;
        float imaginary;
}

struct DOUBLE COMPLEX {
        double real;
        double imaginary;
}
```

I.1.1 F77L Main Programs

An F77L program can call external functions written in Turbo C; however, C does not check to verify that the proper number of arguments are supplied. The programmer should be particularly careful to pass the proper arguments. If this is not done, the program may abort or exhibit other bizarre behavior.

Turbo C modules called from F77L can use functions from the Turbo C run-time math library if you include the Lahey-supplied library "BCMATHL.LIB" in the link cammand. This library must come before the Borland "MATHL.LIB" Library. For example:

```
...,FP87,BCMATHL+MATHL+CL+F77L;
```

The "BCMATHL" Library causes some F77L intrinsics to be used in place of those Turbo C functions not directly supported.

NOTE: The "BCMATHL" Library is not required to use Turbo C's Add, Subtract, Multiply, and Divide operations.

The BC EXTERNAL statement allows a programmer to call routines which use Turbo C calling conventions. The syntax of the BC EXTERNAL statement is:

```
BC EXTERNAL sub[,sub]...
```

Where:

"sub" is the name of a subroutine or function that is to be invoked using Turbo C conventions.

Example:

```
BC EXTERNAL fun1
WRITE(*,*) fun1 (100, 200)
```

"fun1" will be invoked like a Turbo C Float (REAL*4) Function.

Values returned from the Turbo C functions will operate correctly in an F77L function context if the type of the F77L name agrees with the type of the C function. INTEGER*2, INTEGER*4, REAL*4, and REAL*8 are the only function types supported.

To use a library of routines in this manner, we suggest that you create an include file containing BC EXTERNAL statements for all of the names of the routines you use from that library. If the routines are to be invoked as functions, also designate the type of the routine name. Then INCLUDE that file in each program unit that invokes any of the library routines.

BC EXTERNAL still acts like the EXTERNAL statement in that it makes the specified routine name known to the linker, which will search for that name in other "*.OBJ" or "*.LIB" files.

The system function CARG has been provided to pass values to C. CARG causes F77L CHARACTER Arguments to be passed like C Character Strings (the argument is copied to the stack, and a NULL byte is appended). If an INTEGER, INTEGER*2, REAL variable or constant appears in the CARG function, then F77L PUSHes its value instead of its address onto the stack. CARG can only be used in an argument list and can only pass values to a subprogram; nothing can be returned to a CARG function argument.

The following table summarizes the data types that can be passed from F77L subprograms to C functions. The "—" indicates arguments types that CANNOT BE PASSED.

FORTRAN Type	C Type
INTEGER*2	int *
CARG(INTEGER*2)	int
INTEGER*4	long int *
CARG(INTEGER*4)	long int
REAL*4	float *
CARG(REAL*4)	float
DOUBLE PRECISION	double *
COMPLEX	float [2]
COMPLEX*16	double [2]
LOGICAL*1	char *
LOGICAL*4	—
CHARACTER	char[ ]
(for the above, the F77L program must append a NULL byte to terminate the string).	
CARG(CHARACTER)	char[ ]
label	—
EXTERNAL	—

Linking in "F77LBC.OBJ" also allows Turbo C functions to use C input/output, memory functions, etc. When writing to a file or device in a C function, you must flush the buffers explicitly by using FFLUSH, FCLOSE, etc., in the C code before terminating execution.

Choosing the proper libraries for linking depends upon what switches were used to compile your main program. For programs compiled with /NE or /E/2, use

```
OPTLINK F77LBC+FPROG+CPROG,FPROG,nul,FP87+MATHL+CL+F77L;
```

For programs compiled with /E but without /2, replace "FP87.LIB" with "EMU.LIB".

Stack Overflow Protection

When an F77L main program is used, dynamic stack management is performed on entry to and exit from each F77L subprogram. Any non-F77L routine called by an F77L subprogram has a minimum of 240 bytes of stack storage available for its use. If the C-compiled functions require more than 240 bytes of stack storage (including any required by DOS functions invoked by the C functions), then additional stack storage overhead can be set by calling the subroutine ENSSTK. This routine only needs to be called once each time the program is executed. The argument passed to it becomes the number of overhead stack storage bytes reserved for non-F77L routines called by F77L routines (instead of 240). ENSSTK may be called from FORTRAN:

```
CALL ENSSTK (1000)
```

or from C:

```
int stk = 1000;
ensstk (&stk);
```

Program example:

```
        PROGRAM BC
        BC EXTERNAL CFUN
        INTEGER I/1000/
        COMPLEX CX /(3.5,4.5)/
        CHARACTER*11 A*12, C/"Hello there"/
        CALL CFUN (I, CX, CARG(C), A)
*       Print A up to the null byte
        PRINT *,A(1:INDEX(A,CHAR(0)) – 1)
        END

        #include "stdio.h"
        #include "string.h"
        cfun (i, cx, c, a)
        int *i;
        float cx[2];
        char *c;
         char *a;
        {

              FILE *out;
              out = fopen ("CON","wa");
              fprintf (out, "%d (%e, %e) %s\n", *i,
              cx[0],cx[1],c);
              fclose (out);
              /* copy input string back to F77L */
              strcpy (a, c);
        }
```

I.1.2 Turbo C Main Programs

Turbo C main programs may call F77L subroutines and functions with most features of both languages functioning normally. The SYSTEM subroutine and the CHAIN statement are not usable in this context. Note that the F77L subroutines OVEFL, UNDFL, DVCHK, and INVALOP are usable, but they will mask their respective error conditions for the remainder of the entire program, including any remaining C code. The names of any C subroutines or functions that are to be called by FORTRAN must appear in a BC EXTERNAL statement. This causes F77L to prepend an underline character to the program unit name. ENTRY statements cannot be used in this context.

The F77L program units and the C program units must be compiled according to the rules listed earlier in this appendix, and all C arguments must be pointers. Numeric or CHARACTER constants may not appear in C argument lists (string constants may). Also, any files opened in FORTRAN must be explicitly closed before execution is terminated.

The F77L-provided interface module "BCF77L.OBJ" must be linked with the Turbo C initialization module, "C0L.OBJ", and the C and F77L libraries. NDP emulation with Turbo C will work if you compile your FORTRAN subroutines with the /E switch and without /2. Choosing the appropriate Turbo C math libraries is the same as explained in the previous section. For example, if the C main function is "CM.OBJ", the F77L subroutines are in "F77LSUBS.OBJ" and were compiled with /E, then the OPTLINK Command would be:

```
OPTLINK BCF77L+C0L+CM+F77LSUBS,CMAIN,,EMU+MATHL+CL+F77L;
```

Using this method the F77L subroutines will do stack checking like Turbo C instead of the usual F77L method. Errors will be diagnosed by F77L and reported using traceback (until a call from a C-compiled program unit) just as they would in an F77L environment. Input/output can be performed in both languages (as long as the same file is not open in both languages), and all F77L functions and system subroutines will execute correctly, except for the previously stated exceptions.

Most data types can be passed from Turbo C functions to F77L subprograms. The following table summarizes the relationships between the C and F77L types.

C Type	**FORTRAN Type**
int *	INTEGER*2
long int *	INTEGER*4
float *	REAL*4
double *	DOUBLE PRECISION
float [2]	COMPLEX
double [2]	COMPLEX*16
char *	LOGICAL*1
struct CHARACTER { char *text; int length;} *	CHARACTER *(*)

F77L requires a 4-byte pointer for CHARACTER arguments. This pointer addresses a 6-byte structure that consists of a 4-byte pointer followed by 2-byte length. Program example:

```
/* Structure for passing characters to F77L */
struct CHARACTER {
        char *text;
        int length;
        };
main () {
        long i = 100L;
        static char msg[] = {"Hello there"};
        struct CHARACTER c;
        static double cx[2] = {3.0, 4.0};
        ...
/* Put the pointer & length into the structure */
        c.text = msg;
        c.length = strlen(msg);
        fortsub (&i, &c, cx);
        }
SUBROUTINE FORTSUB (I, C, CX)
        BC EXTERNAL FORTSUB
        INTEGER I
        CHARACTER *(*) C
        COMPLEX *16 CX
        PRINT *,I,C,CX
        END
```

I.2 Microsoft C Interface

Different versions of Microsoft C require different "F77LMSC.OBJ" modules. The names of the modules are of the form F77LMSC.###, where ### corresponds to the Microsoft C version number (e.g., 400 for version 4.00). If your version number is not included in a module name, use the highest version number below your version number.

NOTE: Microsoft C main programs may not call F77L-compiled subprograms directly. In order to call F77L subprograms from Microsoft C, a simple F77L main program must be created to call the C main program as if it were a subprogram. Once the C main program has been loaded as a subprogram, it may then call F77L-compiled subprograms if the conventions explained in this appendix are followed.

You can either choose to rename the version-specific modules to object files, e.g.

ren *.400 *.obj

or leave the distribution extensions intact, and specify the entire module name at link time.

To compile your Microsoft C source use:

CL /Alfw /Gs /c filename

at the DOS prompt.

To compile FORTRAN source, use:

F77L filename/NI

plus any other option switches desired.

Link the program using both "F77L.LIB" and the Microsoft C Library(s) for large data and code. Always specify the F77L Library after all other libraries in your link command. See the examples in the following sections.

Suggestion: Whenever possible, test your C functions in a C-only environment and your FORTRAN routines in an F77L-only environment before mixing environments. This should make debugging easier.

In addition to the above restrictions, programmers must be particularly careful when passing arrays between FORTRAN and C. Only one-dimensional arrays can easily be passed to C. For example, a C subprogram called from FORTRAN with an INTEGER array of ten elements could be written:

```
cfunc(int_array)
long int int_array[10];
{
...
```

where the actual size of the passed array will not be checked to verify that it is ten elements long; a second argument is often passed, giving the actual size of the array.

Since FORTRAN stores multidimensional arrays in column-major order, while C stores arrays in row-major order, it is usually not convenient to pass multidimensional arrays to non-FORTRAN routines. If this is done, the element of an n-dimensional array "A" that is declared:

```
A(I1,I2,...,In)
```

in FORTRAN must be accessed with subscripts REVERSED in C, for example:

```
A[In][In–1][...][I1]
```

Also, the first element of a C array dimension is 0, so FORTRAN arrays should be dimensioned (0:n–1) for an array of size n.

NOTES: F77L functions can be called by Microsoft C-compiled code if the first C argument is a pointer where the F77L function value is to be stored.

The names of any C subroutines or functions that are to be called by FORTRAN must appear in a MSC EXTERNAL statement. This causes F77L to prepend an underline character to the program unit name. ENTRY statements cannot be used in this context.

FORTRAN files cannot be directly accessed from other languages, since FORTRAN refers to files using unit numbers, and other languages have a file data type. The general rule is: a given file can be operated on either with FORTRAN code or non-FORTRAN code, but not both. For instance, to do input/output with a C-keyed file from FORTRAN, C routines will have to be called to open the file, add and delete records, etc.

The following structures may be useful for working with F77L data types in C:

```
struct CHARACTER {
        char *text;
        int length;
}

struct COMPLEX {
        float real;
        float imaginary;
}

struct DOUBLE COMPLEX {
        double real;
        double imaginary;
}
```

I.2.1 F77L Main Programs

An F77L program can call external functions written in Microsoft C; however, C does not check to verify that the proper number of arguments are supplied. The programmer should be particularly careful to pass the proper arguments. If this is not done, the program may abort or exhibit other bizarre behavior.

The MSC EXTERNAL statement allows a programmer to call routines which use Microsoft C calling conventions. The syntax of the MSC EXTERNAL statement is:

```
MSC EXTERNAL sub[,sub]...
```

Where:

> "sub" is the name of a subroutine or function that is to be invoked using Microsoft C conventions.

Example:

```
MSC EXTERNAL fun1
WRITE(*,*) fun1 (100, 200)
```

> "fun1" will be invoked like a Microsoft C Float (REAL*4) Function.

Values returned from the Microsoft C functions will operate correctly in an F77L function context if the type of the F77L name agrees with the type of the C function. INTEGER*2, INTEGER*4, REAL*4, and REAL*8 are the only function types supported.

To use a library of routines in this manner, we suggest that you create an include file containing MSC EXTERNAL statements for all of the names of the routines you use from that library. If the routines are to be invoked as functions, also designate the type of the routine name. Then INCLUDE that file in each program unit that invokes any of the library routines.

MSC EXTERNAL still acts like the EXTERNAL statement in that it makes the specified routine name known to the linker, which will search for that name in other "*.OBJ" or "*.LIB" files.

The system function CARG has been provided to pass values to C. CARG causes F77L CHARACTER Arguments to be passed like C Character Strings (the argument is copied to the stack, and a NULL byte is appended). If an INTEGER, INTEGER*2, REAL variable or constant appears in the CARG function, then F77L PUSHes its value instead of its address onto the stack. CARG can only be used in an argument list and can only pass values to a subprogram; nothing can be returned to a CARG function argument.

The following table summarizes the data types that can be passed from F77L subprograms to C functions. The "—" indicates arguments types that CANNOT BE PASSED.

FORTRAN Type	**C Type**
INTEGER*2	int *
CARG(INTEGER*2)	int
INTEGER*4	long int *
CARG(INTEGER*4)	long int
REAL*4	float *
CARG(REAL*4)	float
DOUBLE PRECISION	double *
COMPLEX	float [2]
COMPLEX*16	double [2]
LOGICAL*1	char *
LOGICAL*4	—
CHARACTER	char[ ]
(for the above, the F77L program must append a NULL byte to terminate the string).	
CARG(CHARACTER)	char[ ]
label	—
EXTERNAL	—

Linking in "F77LMSC.OBJ" also allows Microsoft C functions to use C input/output, memory functions, etc. When writing to a file or device in a C function, you must flush the buffers explicitly by using FFLUSH, FCLOSE, etc., in the C code before terminating execution.

Microsoft C Version 4.X Example:

```
OPTLINK  F77LMSC+FPROG+CPROG,FPROG,nul,87+LLIBFP+LLIBC+LIBH+F77L;
```

If /E was used to compile your main program, replace "87.LIB" with "EM.LIB".

Microsoft C Version 5.X Example:

```
OPTLINK  F77LMSC+FPROG+CPROG,FPROG,nul,LLIBC7+F77L/NOE;
```

If /E was used to compile your main program, replace "LLIBC7.LIB" with "LLIBCE.LIB".

Stack Overflow Protection

When an F77L main program is used, dynamic stack management is performed on entry to and exit from each F77L subprogram. Any non-F77L routine called by an F77L subprogram has a minimum of 240 bytes of stack storage available for its use. If the C-compiled functions require more than 240 bytes of stack storage (including any required by DOS functions invoked by the C functions), then additional stack storage overhead can be set by calling the subroutine ENSSTK. This routine only needs to be called once each time the program is executed. The argument passed to it becomes the number of overhead stack storage bytes reserved for non-F77L routines called by F77L routines (instead of 240). ENSSTK may be called from FORTRAN:

```
CALL ENSSTK (1000)
```

or from C:

```
int stk = 1000;
ensstk (&stk);
```

Program example:

```
          PROGRAM MSC
          MSC EXTERNAL CFUN
          INTEGER*2 I/1000/
          COMPLEX CX /(3.5,4.5)/
          CHARACTER*11 A*12, C/"Hello there"/
          CALL CFUN (I, CX, CARG(C), A)
*         Print A up to the null byte
          PRINT *,A(1:INDEX(A,CHAR(0)) – 1)
          END

          #include "stdio.h"
          #include "string.h"
          cfun (i, cx, c, a)
          int *i;
          float cx[2];
          char *c;
          char *a;
          {
                FILE *out;
                out = fopen ("CON","wa");
                fprintf (out, "%d (%e, %e) %s\n", *i,
                cx[0],cx[1],c);
                fclose (out);
                /* copy input string back to F77L */
                strcpy (a, c);
          }
```

I.3 Lattice C Interface

Different versions of Lattice C require different "F77LLC.OBJ" and "LCF77L.OBJ" modules. The names of the modules are of the form F77LLC.### and LCF77L.###, where ### corresponds to the Lattice C version number (e.g., 200 for version 2.00 and 215 for version 2.15). If your version number is not included in a module name, use the highest version number below your version number.

NOTE: F77L is compatible with Lattice C versions 2.15 to 3.21. Lattice C versions 3.22 or later are not supported.

If your Lattice C version is 2.15, use the following DOS command to get the files named correctly:

REN *.215 *.OBJ

To compile the Lattice C Version 2.15 Source, use:

LC1 filename -mL -s
LC2 filename -v

If your Lattice C version is 3.00 through 3.21, use the following DOS command to get the files named correctly:

REN *.300 *.OBJ

To compile the C version 3.00 source, use:

LC1 -mLs (filename)
LC2 -v(filename)

NOTE: See your Lattice C version's argument conventions.

To compile FORTRAN source, use:

F77L filename/NI

Link the program using both "F77L.LIB" and the Lattice C Library, "LC.LIB", for large data and code. Always specify the F77L Library after all other libraries in your OPTLINK Command. See the examples in the following sections.

Emulation is not supported in F77L subroutines if the main program is a C program compiled with Lattice C. In order to support emulation, you must create a simple F77L main program (compiled with /E) to call the Lattice C main program. Once the Lattice C main program has been loaded as a subprogram, it may then call F77L subprograms that were compiled with emulation.

In programs where the main module is in C, do not call a C function from an F77L subprogram that was called by a C function.

Suggestion: Whenever possible, test your C functions in a C-only environment and your FORTRAN routines in an F77L-only environment before mixing environments. This should make debugging easier.

In addition to the above restrictions, programmers must be particularly careful when passing arrays between FORTRAN and C. Only one-dimensional arrays can easily be passed to C. For example, a C subprogram called from FORTRAN with an INTEGER array of ten elements could be written:

```
cfunc(int_array)
long int int_array[10];
{
...
```

where the actual size of the passed array will not be checked to verify that it is ten elements long; a second argument is often passed, giving the actual size of the array.

Since FORTRAN stores multidimensional arrays in column-major order, while C stores arrays in row-major order, it is usually not convenient to pass multidimensional arrays to non-FORTRAN routines. If this is done, the element of an n-dimension array "A" that is declared:

    A(I1,I2,...,In)

in FORTRAN must be accessed with subscripts REVERSED in C, for example:

    A[In][In–1][...][I1]

Also, the first element of a C Array Dimension is 0, so FORTRAN Arrays should be dimensioned (0:n–1) for an array of size n.

NOTES: F77L functions can be called by Lattice C-compiled code if the first C argument is where the F77L function value is to be stored.

FORTRAN files cannot be directly accessed from other languages, since FORTRAN refers to files using unit numbers and other languages have a file data type. The general rule is: a given file can be operated on either with FORTRAN code or non-FORTRAN code, but not both. For instance, to do input/output with a C-keyed file from FORTRAN, C routines will have to be called to open the file, add and delete records, etc.

The following structures may be useful for working with F77L data types in C:

```
struct CHARACTER {
          char *text;
          int length;}

struct COMPLEX {
          float real;
  float imaginary;}

struct DOUBLE COMPLEX {
          double real;
  double imaginary;}
```

I.3.1 F77L Main Programs

An F77L program can call external functions written in Lattice C. However, C does not check to verify that the proper number of arguments is supplied. The programmer should be particularly careful to pass the proper arguments. If this is not done, the program may abort or exhibit other bizarre behavior.

The LC EXTERNAL statement allows a programmer to call routines which use Lattice C calling conventions. The syntax of the LC EXTERNAL statement is:

```
LC EXTERNAL sub[,sub]...
```

Where:

"sub" is the name of a subroutine or function that is to be invoked using Lattice C conventions.

Example:

```
LC EXTERNAL fun1
WRITE(*,*) fun1 (100, 200)
```

"fun1" will be invoked like a Lattice C Float (REAL*4) Function.

Values returned from the Lattice C functions will operate correctly in an F77L function context if the type of the F77L name agrees with the type of the C function. INTEGER*2, INTEGER*4, REAL*4 and REAL*8 are the only function types supported.

To use a library of routines in this manner, we suggest that you create an include file containing LC EXTERNAL statements for all of the names of the routines you use from that library. If the routines are to be invoked as functions, also designate the type of the routine name. Then INCLUDE that file in each program unit that invokes any of the library routines.

LC EXTERNAL still acts like the EXTERNAL statement in that it makes the specified routine name known to the linker, which will search for that name in other "*.OBJ" or "*.LIB" files.

NOTE: Although F77L preserves all characters of any EXTERNAL name, the default action for the Lattice C Compiler truncates any name to eight characters. We suggest compiling your C code with the "-n" option to preserve long names. Be aware that you may need to truncate names in your F77L code when referencing previously compiled C Libraries.

The system function CARG has been provided to pass values to C. CARG causes F77L CHARACTER Arguments to be passed like C Character Strings (the argument is copied to the stack, and a NULL byte is appended). If an INTEGER, INTEGER*2, or REAL variable or constant appears in the CARG function, then F77L PUSHes its value instead of a pointer to it onto the stack. CARG can only be used in an argument list and can only pass values to a subprogram; nothing can be returned to a CARG function argument.

The following table summarizes the data types that can be passed from F77L subprograms to C functions. The "—" indicates arguments types that CANNOT BE PASSED.

FORTRAN Type	C Type
INTEGER*2	int *
CARG(INTEGER*2)	int
INTEGER*4	long int *
CARG(INTEGER*4)	long int
REAL*4	float *
CARG(REAL*4)	float
DOUBLE PRECISION	double *
COMPLEX	float [2]
COMPLEX*16	double [2]
LOGICAL*1	char *
LOGICAL*4	—
CHARACTER	char[ ]
(for the above, the F77L program must append a NULL byte to terminate the string).	
CARG(CHARACTER)	char[ ]
label	—
EXTERNAL	—

Linking in "F77LLC.OBJ" also allows Lattice C functions to use C input/output, memory functions, etc. When writing to a file or device in a C function, you must flush the buffers explicitly by using FFLUSH, FCLOSE, etc., in the C code before terminating execution.

Example:

```
OPTLINK  F77LLC+FPROG+CPROG,FPROG,nul,LC+F77L;
```

Stack Overflow Protection

When an F77L main program is used, dynamic stack management is performed on entry to and exit from each F77L subprogram. Any non F77L routine called by an F77L subprogram has a minimum of 240 bytes of stack storage available for its use. If the C-compiled functions require more than 240 bytes of stack storage (including any required by DOS functions invoked by the C functions), then additional stack storage overhead can be set by calling the subroutine ENSSTK.

This routine only needs to be called once each time the program is executed. The argument passed to it becomes the number of overhead stack storage bytes reserved for non F77L routines called by F77L routines (instead of 240). ENSSTK may be called from FORTRAN:

```
CALL ENSSTK (1000)
```

or from C:

```
int stk = 1000;
ensstk (&stk);
```

Program example:

```
      PROGRAM LC
      LC EXTERNAL CFUN
      INTEGER*2 I/1000/
      COMPLEX CX /(3.5,4.5)/
      CHARACTER*11 A*12, C/"Hello there"/
      CALL CFUN (I, CX, CARG(C), A)
*     Print A up to the null byte
      PRINT *,A(1:INDEX(A,CHAR(0)) – 1)
      END

#include "stdio.h"
#include "string.h"
cfun (i, cx, c, a)
int *i;
float cx[2];
char *c;
char *a;
{
        FILE *out;
        out = fopen ("CON","wa");
        fprintf (out, "%d (%e, %e) %s\n", *i,
        cx[0],cx[1],c);
        fclose (out);
        /* copy input string back to F77L */
        strcpy (a, c);
}
```

I.3.2 Lattice C Main Programs

Lattice C main programs may call F77L subroutines and functions with most features of both languages functioning normally. The SYSTEM subroutine and the CHAIN statement are not usable in this context. Note that the F77L subroutines OVEFL, UNDFL, DVCHK, and INVALOP are usable, but they will mask their respective error conditions for the remainder of the entire program, including any remaining C code.

The F77L program units and the C program units must be compiled according to the rules listed earlier in this appendix, and all C arguments must be pointers. Numeric or CHARACTER constants may not appear in C argument lists (string constants may). Also, any files opened in FORTRAN must be explicitly closed before execution is terminated.

The F77L provided interface module "LCF77L.OBJ" must be linked with the Lattice C initialization module, "C.OBJ", and the C and F77L libraries. For example, if the C main function is "CM.OBJ" and the F77L subroutines are in "F77LSUBS.OBJ", then the OPTLINK Command would be:

```
OPTLINK LCF77L+C+CM+F77LSUBS,CM,,LC+F77L;
```

Using this method the F77L subroutines will do stack checking like Lattice C instead of the usual F77L method. Errors will be diagnosed by F77L and reported using traceback (until a call from a C-compiled program unit) just as they would in an F77L environment. Input/output can be performed in both languages (as long as the same file is not open in both languages), and all F77L functions and system subroutines will execute correctly, except for the previously stated exceptions.

Most data types can be passed from Lattice C functions to F77L subprograms. The following table summarizes the relationships between the C and F77L types.

C Type	FORTRAN Type
int *	INTEGER*2
long int *	INTEGER*4
float *	REAL*4
double *	DOUBLE PRECISION
float [2]	COMPLEX
double [2]	COMPLEX*16
char *	LOGICAL*1
struct CHARACTER { char *text; int length;} *	CHARACTER *(*)

F77L requires a 4-byte pointer for CHARACTER arguments. This pointer addresses a 6-byte structure that consists of a 4-byte pointer followed by 2-byte length.

Program example:

```
/* Structure for passing characters to F77L */
struct CHARACTER {
        char *text;
        int length;
        };
main () {
        long i = 100L;
        static char msg[] = {"Hello there"};
        struct CHARACTER c;
        static double cx[2] = {3.0, 4.0};
        ...
/* Put the pointer & length into the structure */
        c.text = msg;
        c.length = strlen(msg);
        fortsub (&i, &c, cx);
        }

SUBROUTINE FORTSUB (I, C, CX)
        INTEGER I
        CHARACTER *(*) C
        COMPLEX *16 CX
        PRINT *,I,C,CX
        END
```

J ASSEMBLY LANGUAGE INTERFACE

This is a definition of the F77L Execution Environment and a description of how to write assembly language programs to interface with F77L. The sections in this appendix provide details on assembly language interface.

Required Reading

CODE and DATA Segments;

Argument Lists;

Common Blocks;

Assembly Language Interface;

Additional Features; and

Assembly Language Programming Guidelines.

NOTES: Microsoft Macro Assembler Version 5.0 or higher is required.

For an example of an assembly language interface, see the "EXAMPLE.ASM" File on the F77L distribution disks.

J.1 Required Reading

Execution Time Conventions

The following conventions must be observed by the interfacing code:

All arguments passed from F77L-compiled routines are 4-byte pointers (addresses of values). A pointer consists of the OFFSET followed (at a higher address) by the SEGMENT.

All EXTRN symbols and RETs are FAR.

The caller POPs the argument pointers off the stack.

Registers and the six bytes addressed at [BP-6] may be used by the assembly language routine. The Registers (BP, SP, SS and CS) MUST be restored to their original values before the routine exits.

The direction flag will be cleared when F77L calls a subroutine. An assembly subroutine must ensure that it is cleared upon exit (use the CLD instruction).

The NDP is maintained in the round-to-nearest state and is empty when non-intrinsic routines are called. Its stack should be empty when control is returned. An FWAIT instruction must be executed before returning to F77L if the NDP is used.

Segment Registers DS and ES are undefined upon entry to the assembly-written routine.

Unless otherwise specified, all F77L-supplied routines are considered to destroy all registers except CS, IP, SS, SP and BP.

J.2 CODE and DATA Segments

To guarantee user-written assembly language routines and F77L-compiled subprogram units will be linked correctly, segments should be defined as follows (where XXX, Y1 and Z1 are user-defined names):

```
DGROUP   GROUP  DATA
DATA     SEGMENT       WORD      PUBLIC 'DATA'
ASSUME   DS:DGROUP
         ...
DATA     ENDS
XXX      SEGMENT BYTE
ASSUME   CS:XXX
PUBLIC   Y1,Z1
Y1        PROC   FAR
         ...
Y1       ENDP
Z1       PROC    FAR
         ...
Z1       ENDP
XXX      ENDS
END
```

NOTES: Assembly routines already written using different names should link correctly if the class name for the data is "DATA" and the class name for the code is "CODE". Any segments defined in DGROUP must be named "DATA".

If you are going to be addressing DATA in DGROUP, then DS must be set to DGROUP. For example:

```
MOV      AX,DGROUP
MOV      DS,AX          ;set DS accordingly to reference data group
```

J.3 Argument Lists

A 4-byte pointer is PUSHed (SEGMENT is PUSHed before OFFSET) for each non-ALTERNATE RETURN specifier, in a right to left order.

Example:

```
CALL SUB(A, B, ..., N)
```

is compiled as

```
PUSH    pointer to N
...
PUSH    pointer to B
PUSH    pointer to A
```

If a function is invoked, an additional pointer to address the memory where the function value is to be stored is PUSHed.

Example:

```
Y = F(A, B, ..., N)
```

is compiled as

```
PUSH    pointer to N
...
PUSH    pointer to B
PUSH    pointer to A
PUSH    pointer to store F value
```

These pointers address all data types except CHARACTER. For CHARACTER arguments, the pointer addresses not the value but a 6-byte structure which consists of a pointer to the CHARACTER value and the length of CHARACTER item:

```
CHAR_ARG_POINTER equ $
            DW                  OFFSET
            DW                  SEGMENT
            DW                  LENGTH
```

J.4 Common Blocks

A single common block may be divided into several "segments" to be compatible with linkers and the computer hardware. F77L appends an "@" character and a two-digit number to the end of a COMMON name to create the segment names. The first segment is @00, the second @01, and so forth, each having a maximum length of 32768 bytes. If the length of a common block name exceeds 28 characters, any characters beyond 28 are truncated in order to form the segment names.

An assembly language program should only need to reference the first segment as long as adjustments to the segment register are made for crossing boundaries of 65536 bytes.

To reference data in the common, /ARCHIE/, define the first segment as in this example:

```
ARCHIE@00 SEGMENT PARA COMMON 'F77LCOMN'
```

To reference blank common use:

```
_COMMON_@00 SEGMENT PARA COMMON 'F77LBLNK'
```

J.5 Assembly Language Interface

This section describes a simple interface to assembly language which should be sufficient for most applications.

Argument Addressing

The arguments have been PUSHed on the stack (see **Required Reading**) so that the first argument in the argument list is closest to the current stack pointer, immediately following the return address. The BP Register may be used to address the arguments.

Example:

```
ARG_OFFSET EQU  6;                          Offset of first
;                                           argument
SUB        PROC  FAR
           PUSH  BP;                        Save caller's BP
           MOV   BP,SP;                     Address args
; Point to 1st
           LES   BX,DWORD PTR[BP].ARG_OFFSET
           MOV   AX,ES:[BX];                Load 1st argument
; Point to 2nd
           LES   BX,DWORD PTR[BP+4].ARG_OFFSET
           SUB   AX,ES:[BX];                Subtract 2nd argument
; Point to 3rd
           LES   BX,DWORD PTR[BP+8].ARG_OFFSET
           MOV   ES:[BX],AX;                Result to 3rd argument
           POP   BP;                        Recover caller's BP
           RET;                             FAR Return
SUB        ENDP
```

Local Storage

The assembly language programmer can allocate temporary storage on the stack for use by his routine by subtracting from SP, for example:

```
SLEN       EQU   20;                        Amount of storage wanted
SUB        PROC  FAR
           PUSH  BP;                        Save caller's BP
           MOV   BP,SP;                     Address args
           SUB   SP,SLEN;                   Allocate local
; The following line stores into 1st byte of local storage.
           MOV   BYTE PTR [BP-SLEN],3
           MOV   SP,BP;                     Deallocate local storage
           POP   BP;                        Recover caller's BP
           RET;                             FAR return
SUB        ENDP
```

Stack Overflow Protection

Stack overflow checking is performed on entry to, and exit from, each F77L subprogram. Any non-F77L routine called by an F77L subprogram has a minimum of 240 bytes of stack storage available for its use. If an assembly language-written routine requires more than 240 bytes of stack storage (including any required by DOS functions invoked by the routine), then additional stack storage must be reserved by calling "_ENSSTK".

Example:

```
SLEN    EQU     2000;           Amount of storage wanted
        EXTRN   _ENSSTK:FAR
SUB     PROC    FAR
        MOV     AX,SLEN;        Make sure it's available
        CALL    _ENSSTK
        PUSH    BP;             Save caller's BP
        MOV     BP,SP;           Address args
        SUB     SP,SLEN;        Allocate local
```

The FORTRAN callable subroutine ENSSTK may also be used to ensure enough stack storage. This routine only needs to be called once each time the program is executed. The argument passed to it becomes the number of overhead stack storage bytes reserved for non-F77L routines called by F77L routines (instead of 240).

J.6 Additional Features

The execution environment provides many frequently required functions which may be useful to the assembly language programmer. The following descriptions are organized into five sections: **Error Linkages**, **Argument Checking and Alternate Returns**, **NDP Support**, **Additional Subroutines**, and **Calling Intrinsic Functions**.

J.6.1 Error Linkages

Two subroutines are provided to preserve and restore the stack parameters that allow error messages for arithmetic operations to be reported with a traceback. They should be used whenever the assembly-language routine is capable of generating an 8086 or 8088 divide-by-zero, any NDP error, or if the routine calls any intrinsic functions.

_FUNENTR

_FUNENTR saves SP and BP if they have not currently been saved and sets an internal flag for the error routines. This routine should be called at the beginning of the assembly-language routine.

NOTE: The DS Register points to DGROUP after this routine is executed, and the CX Register is destroyed.

_FUNEXIT

_FUNEXIT restores BP and clears the internal flag to indicate to _FUNENTR that SP and BP are no longer saved. The CX and DS Registers have been modified.

Example:

```
        EXTRN   _FUNENTR:FAR
SUBR    PROC    FAR
        CALL    _FUNENTR
        MOV     BP,SP;              Address args
        ...
        CALL    _FUNEXIT
        RET;                        FAR Return
SUBR    ENDP
```

J.6.2 Argument Checking and Alternate Returns

The /I (INTERFACE) Compiler Option - (See the **Compiler Options and Configuring** Section in **Appendix A**) causes code to be generated varying with whether a function or subroutine subprogram is being invoked. The following flags and registers are defined for calling a subroutine after the arguments' addresses (if any) are PUSHed:

```
MOV     AX,n;           Number of args
XOR     BX,BX;          ALT RETURNs
CALL    SUBROUTN;       FAR CALL
ADD     SP,4*n;         Restore STACK
```

NOTES: The XOR does three things which are required by the calling sequence:

CARRY FLAG is cleared, indicating argument checking is in effect to the called subprogram;

ZERO FLAG is set, indicating a subroutine is being invoked;

The number of alternate return specifiers in the subroutine argument list is put into BX. If such specifiers are part of the argument list, then

```
MOV     BX,# of alternate returns
```

is generated after the 'XOR BX,BX'.

The called subroutine returns the number of alternate return specifiers in BX when control returns to the calling subprogram.

If a function subprogram is invoked, then

```
XOR     AX,AX;          Clear CARRY
ADD     AX,n + 1;       Set NZ and n + 1
CALL    FUNCTION;       FAR CALL
ADD     SP,4*(n + 1);   Restore stack
```

is generated to accomplish the following: clear the CARRY FLAG, clear the ZERO FLAG (to nonZERO) and set AX to the number of argument addresses PUSHed. The stack is restored after control is returned from the function.

The "/NI" compiler option causes the compiler to generate:

```
            ...
; PUSH ARGS
            STC
            MOV         AX,# of args
            CALL        SUBPROGRAM
```

J.6.3 NDP Support

In addition to processing NDP interrupts and publishing the appropriate error messages, the runtime package also provides subroutines to ease NDP usage:

_87ROUND

Sets the NDP Control Word to "round to nearest" (sets RC to 00).

_87CHOP

Sets the NDP Control Word to "chop" (sets RC to 11).

_87INIT

Initializes the NDP to empty, round to nearest and interrupt on all exceptions but precision.

_87TEST

Compare the top of the 8086 stack with zero, return the NDP status word in BP-2, and set the 8086 Z and S Flags appropriately. _87TEST explicitly clears the 0 Flag. (These two flags are set as for an 8086 comparision.) Note: the CARRY Flag is not defined and all registers are preserved.

_87TESTP

Test and pop the NDP stack. Note: Preserves all registers.

_87FLAGS

Return the NDP status word in BP-2, and set the 8086 Z and S Flags as if an 8086 comparision had been performed. Note: the CARRY Flag is not defined and all registers are preserved.

J.6.4 Additional Subroutines

The following routines may also be useful to Assembly Language Programmers.

_LONG_SEG

This routine converts a long integer in DX:AX into a SEGMENT in DX and an OFFSET in AX. Note: Destroys CX Register.

_SEG_LONG

This routine converts a SEGMENT in DX and an OFFSET in AX into a long integer in DX:AX. Note: Destroys CX Register.

The following routines, _GETML and _RLSML, use C calling conventions. The input arguments are PUSHed onto the stack, and any result returns in DX and AX. All registers except CS, IP, SS, SP and BP are considered destroyed upon return. Be sure to POP any arguments off the stack after these routines are called.

_GETML

This routine gets a memory block from the "heap". The amount to get is a double word and must be PUSHed onto the stack (PUSH high-order word first) before the routine is called. The address of the memory block returns in DX (SEGMENT) and AX (OFFSET), unless a large enough block is not available, in which case DX and AX return with the value 0.

_RLSML

This routine returns a memory block that was gotten by _GETML to the "heap". Before calling _RLSML, PUSH the following onto the stack: amount to be returned (double word — PUSH high-order word first), followed by the SEGMENT and OFFSET of the block. AX returns 0 (zero) if the release was completed, or minus one if the memory block overlaps a block already in the heap.

J.6.5 Calling Intrinsic Functions

This section makes the F77L Intrinsic Functions (contained in the "F77L.LIB") available to Assembly Language Programmers by describing the calling sequences.

The intrinsic functions with numeric arguments and numeric results (e.g., SIN, LOG, ATAN) communicate via the NDP:

```
Y = SIN(X)
```

is compiled as

```
FLD     DWORD PTR X
CALL    @SINA
FSTP    DWORD PTR Y
```

Two argument functions load the second argument before the first (from right to left as stated in the conventions above). For COMPLEX arguments, load the REAL part before the imaginary part.

NOTE: To convert a REAL number that is in the top of the NDP stack to COMPLEX it is only necessary to FLDZ.

Intrinsic functions with CHARACTER arguments or results (e.g., CHAR, LEN, INDEX) use the following conventions: pointers to arguments are PUSHed right to left, the function value is returned in the "zeroth" argument pointer and the caller pops the argument pointers from the stack.

K LAHEY-COMPATIBLE SOFTWARE INTERFACES

F77L has been designed to produce output that is compatible with specific software products authored by other companies. These other software products are called "Lahey-Compatible Products". Lahey maintains lists of compatible products that can be obtained by calling Lahey Computer Systems or using Lahey's BBS system. Call the vendor who supplies the compatible program you require for details regarding an F77L FORTRAN Language System Interface.

K.1 Lahey PLINK Overlay Linker

Lahey Computer Systems distributes a subset of the Phoenix PLINK86 PLUS Overlay Linker called Lahey PLINK. This linker is fully compatible with F77L and allows single-level overlays of source code modules. Lahey PLINK includes the PLIB86 Library Manager to build, modify, index, and merge libraries. To use Lahey PLINK, refer to the Lahey PLINK Reference Manual for detailed instructions. For multi-level overlays or data module overlays, use the full Phoenix PLINK86 PLUS version described below.

K.2 PLINK86 Overlay Linker

F77L programs may be linked using PLINK86. To use PLINK86 to overlay F77L programs, the following are necessary in addition to the overlay structure described in the Phoenix documentation:

```
FILE <ROUTINES FOR THE ROOT>
OVERLAY F77LCODE,F77LDATA
```

The OVERLAY statement will cause F77L-generated code and constant data to be overlaid. To also overlay named common blocks and SAVEd local variables, include F77LCOMN in the OVERLAY statement.

Care must be taken when overlaying data areas, since each time the overlay containing the data is read in from disk, all data are set to their initial values. If most local data does not need to be SAVEd, the compiler option "/NR", which allocates local variables on the stack, is usually more space-efficient than overlaying local data.

The "RELOAD" Command in PLINK86 forces overlays to be reloaded after exiting a routine. We do NOT recommend using this command with F77L because it destroys our error traceback facility and reduces the effectiveness of SOLD.

If you use SOLD on an executable file linked using "reload", be aware of the following:

- The "A" command will not work

- Any Breaks or Traces on "Line", "Change" or "When" can only be for a single program unit at a time, and the program unit may not be overlaid while it is "active" and a Break or Trace is in effect.

K.3 Microsoft FORTRAN Interface

The MS EXTERNAL statement allows a programmer to call routines which use Microsoft FORTRAN (Versions 3.2 and above) calling conventions for functions and subroutines. The interface will work correctly only for routines written in assembly language, and not for routines actually compiled by a Microsoft FORTRAN Compiler, since they require the Microsoft Library and runtime environment. MS EXTERNAL does not correctly pass the length of a character, so library routines cannot get the character length available in Microsoft FORTRAN 4.0.

The syntax of the MS EXTERNAL statement is:

```
MS EXTERNAL sub[,sub]...
```

Where:

"sub" is the name of a subroutine or function that is to be invoked using Microsoft FORTRAN conventions. Example:

```
MS EXTERNAL fun1
WRITE(*,*) fun1 (100, 200)
```

"fun1" will be invoked like a Microsoft (Versions 3.2 and above) FORTRAN function.

To use a library of routines in this manner, we suggest that you create an INCLUDE file containing MS EXTERNAL statements for all of the names of the routines you use from that library. Declare the types of the names of any routines that are functions. Then INCLUDE that file in each program unit that invokes any of the library routines.

While many third party libraries designed to interface with Microsoft FORTRAN will work with F77L using the MS EXTERNAL statement, some may not. Possible reasons for not working are that they reference routines in the Microsoft FORTRAN library, that they use coding conventions which depend upon being linked with a Microsoft main program, or that they have bugs that for some reason did not manifest themselves before.

K.4 Other Lahey-Compatible Software Products

Other software manufacturers have or are developing interfaces specifically for F77L. Their documentation will include information on how to link to F77L programs.

RESERVED FOR YOUR NOTES

ERROR MESSAGES & REFERENCES

ERROR MESSAGES
& REFERENCES

L F77L ERROR MESSAGES

F77L Error Messages have been designed to be self explanatory. The following information explains how to use the messages and probable causes for some of them.

L.1 Compiler Error Messages

Syntax errors and other errors detected while parsing (determining exactly what the statement is) display the source line and an up-arrow (^) pointer to the character at which the error was discovered. Usually if the correction is not immediately obvious, checking the statement syntax in a FORTRAN manual will be all that is required to determine the error. For the few messages that could be puzzling, we include the following information to assist in correcting the error. Error messages are in normal type and their explanations are in *italics.*

"cannot be allocated due to poor declaration"

> *A previous FATAL error message about the named item indicates what is wrong with the declaration.*

"EQUIVALENCE extends a COMMON forward"

> *An EQUIVALENCE statement defines an array that would have to begin at an address lower than the beginning of the common.*

"appeared previously in TYPE/LENGTH context"

> *The length of a CHARACTER item is specified more than once in a program unit.*

"Memory exhausted, increase memory or reduce program unit size"

If this message appears, the compiler is unable to compile the current program unit in the available memory. Giving the compiler more memory by increasing its partition size, removing memory-resident software, etc. should allow the program to compile. If you cannot give the compiler enough memory, try compiling the program unit in a file by itself. If that doesn't help, divide the program unit into two or more subprograms.

"Compiler internal table exceeds 65520 bytes, table no. XX.

Where "XX" identifies one of the following tables. This should happen only in extremely large program units. If the number of the table (XX) is one of the following, the text indicates what FORTRAN entities caused the overflow. In the unlikely event you get some other number, please call Lahey Technical Support.

Table #	***Meaning***
0	*Total identifier literal text exceeds 32767 bytes.*
1	*Number of identifiers exceeds 4095.*
2	*Number of labels exceeds 4680.*
3	*Total numeric constant text exceeds 65520 bytes.*
7	*Total CHARACTER constant text exceeds 65520 bytes.*
10	*Total number of unique substrings exceeds 5460.*
19-22	*One of the DATA statement tables exceeds 65520 bytes. Do the initialization in more than one program unit.*
23	*Total text for EQUIVALENCE statements exceeds 65520 bytes. Call Lahey for technical support.*
26	*Total "Polish" for function definition statements exceeds 65520 bytes.*
49	*More than 7280 lines of code in a program unit.*
59	*More than 65520 bytes of cross-reference information. Reduce number of identifiers, program unit size or use the "/NX" Option switch.*

L.2 Runtime Error Messages

The following table shows the error messages that may occur during execution of an F77L-compiled program and some explanation for messages which are not completely self-explanatory. The error number is the number which is displayed if the error message file (.EER) is not available to the runtime error processor. The IOSTAT number is the number which will be returned for IOSTAT=.

The IOSTAT= specifier may be used with most input/output statements (see Sections 9.1 and 9.14). If an IOSTAT= specifier is present and an error occurs, a value is returned to provide an encoding of the error. Part of this encoding is a number that indicates what error occurred.

The following procedure converts an IOSTAT error code to the equivalent value listed in this table.

```
          OPEN (1,'file1',IOSTAT=errcode,ERR=500)
          ...
    500   PRINT, 'error code =',MOD(errcode, 256)
          ...
```

The value returned by the IOSTAT= specifier is a 16-bit machine word, divided into the two fields:

Bits	Contents
0-7	Error message printing flags
8-15	Error code

The data in bits 0 through 7 are used by the internal FORTRAN runtime routines.

The following table lists the error codes, IOSTAT numbers and the errors to which they refer.

Error Number	Error Message Text
1	Insufficient RAM to Continue Execution
2	Program Stack Exhausted
3	NDP Divide by Zero
4	NDP Arithmetic Overflow
5	NDP Arithmetic Underflow
6	NDP Error - Invalid Number, Integer Overflow, or 0/0 *See **Appendix B**, Section B.2.*
7	Integer Divide Error
8	INTEGER*2 Overflow
9	Chain File Error *File name given in CHAIN statement is invalid, does not exist, or is not an F77L-compiled ".EXE" file.*
10	System Error During Chain *Chain was unable to write either blank common or the FCB's for OPENed files to a temporary file, or memory blocks could not be adjusted.*
no #	System Error After Chain *The Chained-to program was unable to open and read the temporary file passed to it by Chain.*
11	SUBROUTINE Subprogram Invoked as a FUNCTION
12	FUNCTION Subprogram Invoked as a SUBROUTINE
13	Subprogram Argument Count Differs from Caller
14	Subprogram Alternate Return Count Differs from Caller
15	Substring Bounds are Poorly Defined
16	Array Subscript Exceeds Allocated Area
17	Not Used
18	Not Used
19	DO Increment is Zero
20	Adjustable Array Dimension is Not Positive
21	Invalid Argument Value for ABS Function
22	SQRT Argument Negative
23	Invalid Argument Value for MOD Function
24	Invalid Argument Value for INT Function
25	Invalid Argument Value for LOG Function
26	Invalid Argument Value for LOG10 Function
27	Invalid Exponentiation
28	Invalid Exponentiation
29	Invalid Argument Value for SIN Function
30	Invalid Argument Value for COS Function
31	Invalid Argument Value for TAN Function

Error Number	Error Message Text
32	Invalid Argument Value for TANH Function
33	Invalid Argument Value for COSH Function
34	Not Used
35	Invalid Argument Value for ASIN Function
36	Invalid Argument Value for ACOS Function
37	Not Used
38	Invalid Argument Value for ATAN2 Function
39	Not Used
40	COMPLEX Divide by Zero
41	Not Used
42	Runtime System Error in Subroutine "SYSTEM" *An unexpected error occurred while invoking the subroutine SYSTEM.*
43	Insufficient RAM for "COMMAND.COM" in Subroutine "SYSTEM"
44	"COMMAND.COM" Is Not Available in Subroutine "SYSTEM"
45-50	Not Used
51	Invalid Argument Value for COTAN Function
52	COTAN Not Defined for Argument of a Multiple of p
53	Invalid Argument Value for DCOTAN Function
54	DCOTAN Not Defined for Argument of a Multiple of p
55	GAMMA Not Defined for $x < -169.9$ or $x > 170.7$
56	GAMMA Not Defined for Zero or Negative Integer
57	DGAMMA Not Defined for $x < -169.9$ or $x > 170.7$
58	DGAMMA Not Defined for Zero or Negative Integer
59	ALGAMA Not Defined for $x \le 0.0$
60	DLGAMA Not Defined for $x \le 0.0$
61	Invalid Bit Number Specification in BTEST Function
62	Invalid Bit Number Specification in IBSET Function
63	Invalid Bit Number Specification in IBCLR Function
64	Inconsistent Argument(s) in IBITS Function
65	Inconsistent Argument(s) in MVBITS Function
66	Inconsistent Argument(s) in ISHFT Function
67	Inconsistent Argument(s) in ISHFTC Function
68-80	Not Used

The following errors may occur during execution of an input/output statement.

Error No.	IOSTAT Number	Error Message Text
81	1	Insufficient RAM to Continue Execution
82	2	Program Stack Exhausted
83	3	NDP Divide by Zero
84	4	NDP Arithmetic Overflow
85	5	NDP Arithmetic Underflow
86	6	NDP Error - Invalid Number, Integer Overflow, or 0/0 *See **Appendix B**, Section B.2.*
87	7	Integer Divide Error
88	8	INTEGER*2 Overflow
89		Not Used
90		Not Used
91	11	FORMAT Repeat Count Less Than 1 or Greater Than 32767
92	12	Invalid Direct File Record Number. *Record number must be positive.*
93	13	Data Transfer Beyond End of Record
94	14	Data Transfer Beyond End of File
95	15	System I/O Error *An error in the I/O system which was caused by unexpected circumstances.*
96	16	Invalid or Inconsistent Close Parameters *Parameters on the CLOSE statement conflict with those on the OPEN statement.*
97	17	Invalid or Missing Record Length Specification *Record length is not positive. Record length was not specified when required, or was specified for unformatted sequential.*
98	18	Invalid ACCESS= Specification *Specification text not recognized.*
99	19	Invalid or Inconsistent FORM= Specification *Specification text not recognized or conflicts with other specifiers.*
100	20	Invalid STATUS= Specification *Specification text not recognized.*

Error No.	IOSTAT Number	Error Message Text
101	21	BLANK= Specification Invalid or Used with Unformatted File *Specification text not recognized or is used with an unformatted file.*
102	22	Unable to Create File *Disk or directory is probably full.*
103	23	Invalid Parameter Change on Open File *Only CARRIAGE CONTROL=, RECL=, PAD=, DELIM= and BLANK= parameters may be changed when OPENing a file which is already open.*
104	24	Invalid I/O Operation on a non-Disk Device *DOS non-disk devices cannot be backspaced, rewound or closed.*
105	25	Invalid Syntax in FORMAT
106	26	FORMAT Specification Incompatible with Data Type
107	27	Too Many Conversion Digits Requested in FORMAT
108	28	Invalid CARRIAGECONTROL= Specification *Specification text not recognized or is used on a file that is not formatted sequential.*
109	29	Invalid FORMAT Field Width
110	30	Invalid Logical Input
111	31	Invalid Character Input
112	32	Invalid List-Directed Input
113	33	Invalid Numeric Input
114	34	Output Field Width Exceeded
115	35	Invalid Unit Number *Unit numbers must be positive numbers less than 32768.*
116	36	Initiating I/O While Doing I/O *Functions in an I/O list may not themselves perform I/O.*

Error No.	IOSTAT Number	Error Message Text
117	37	Direct I/O on a Sequential File *Attempting to OPEN a file or device that can only be accessed sequentially with ACCESS="DIRECT", or specifying a record number on a sequential file.*
118	38	Sequential I/O on a Direct File *Attempting to OPEN a file that can only be accessed as a direct file with ACCESS= "SEQUENTIAL".*
119	39	Unit Already Connected
120	40	Invalid Hollerith Constant in FORMAT
121	41	File Is in an Inconsistent State *Due to a previous error condition, I/O may not be performed on this unit.*
122	42	Unable to Position File
123	43	Unable to Close File
124	44	Unable to Read File
125	45	Unable to Write File
126	46	Invalid Repeated Input
127	47	List-Directed I/O Not Allowed
128	48	Nesting too Deep in FORMAT
129	49	Conversion Specification Missing in FORMAT
130	50	Invalid Scale Factor in FORMAT
131	51	Unable to Delete File
132	52	File Already in Use
133	53	Formatted I/O on an Unformatted File
134	54	Unformatted I/O on a Formatted File
135	55	File Header is Incompatible with FORM and ACCESS *An unformatted file or formatted direct file must have an F77L Header that describes the file type.*
136	56	File Is Already in the Directory
137	57	Cannot Open File
138	58	Invalid File Name
139	59	No File Connected to Unit
140	60	Invalid Open Parameters

Error No.	IOSTAT Number	Error Message Text
141	61	Too Many Digits of Precision Requested for Data Type *An attempt was made to display more digits of precision than the data type is capable of representing.*
142	62	No File Handles Left (see Configuring, FILES= in DOS Manual)
143	63	Number in FORMAT Is Negative or Too Large *A number in a FORMAT is either negative, greater than 32767, or, if it follows a format specifier, greater than 255.*
144	64	Invalid Value for T or X in FORMAT *Value for TL or TR is negative or greater than 127, or no value was specified for T, or repeat count for X is greater than 127.*
145	65	Path Not Found
146	66	Invalid NAMELIST Input Format
147	67	Array Subscript Exceeds Allocated Area *This error occurs in the NAMELIST function.*
148	68	Wrong Number of Array Dimensions *This error occurs in the NAMELIST function.*
149	69	Variable Name Not Found *This error occurs in the NAMELIST function.*
150	70	File Specified STATUS= "NEW" Already Exists
151	71	File Specified STATUS= "OLD" Does Not Exist
152	72	RECL Specifier Differs from Record Length of File
153	73	Filename Length Exceeds 51 Characters
154	74	Record Length > 32767
155	75	Invalid ACTION= Specification
156	76	Invalid DELIM= Specification
157	77	Invalid PAD= Specification
158	78	Invalid POSITION= Specification
159	79	File Cannot Be Opened *Requested ACTION not available.*
160	80	File-sharing Is Not Loaded *Requested ACTION not available.*

RESERVED FOR YOUR NOTES

M ASCII CHARACTER SET

FORTRAN programs may use the full ASCII Character Set as listed below. The characters are listed in collating sequence from first to last. Characters preceded by up arrows (^) are ASCII Control Characters.

Char	Hex Value	Decimal Value	ASCII Abbr	Comments
^@	00	0	NUL	null
^A	01	1	SOH	start of heading
^B	02	2	STX	start of text
^C	03	3	ETX	break, end of text
^D	04	4	EOT	end of transmission
^E	05	5	ENQ	enquiry
^F	06	6	ACK	acknowledge
^G	07	7	BEL	bell
^H	08	8	BS	backspace
^I	09	9	HT	horizontal tab
^J	0A	10	LF	line feed
^K	0B	11	VT	vertical tab
^L	0C	12	FF	form feed
^M	0D	13	CR	carriage return
^N	0E	14	SO	shift out
^O	0F	15	SI	shift in
^P	10	16	DLE	data link escape
^Q	11	17	DC1	device control 1
^R	12	18	DC2	device control 2
^S	13	19	DC3	device control 3
^T	14	20	DC4	device control 4
^U	15	21	NAK	negative acknowledge
^V	16	22	SYN	synchronous idle
^W	17	23	ETB	end of transmission block
^X	18	24	CAN	cancel
^Y	19	25	EM	end of medium
^Z	1A	26	SUB	end-of-file

Char	Hex Value	Decimal Value	ASCII Abbr	Comments
^[	1B	27	ESC	escape
^\	1C	28	FS	file separator
^]	1D	29	GS	group separator
^^	1E	30	RS	record separator
^	1F	31	US	unit separator
	20	32	SP	space, blank
!	21	33	!	exclamation point
"	22	34	"	quotation mark
#	23	35	#	number sign
$	24	36	$	dollar sign
%	25	37	%	percent sign
&	26	38	&	ampersand
'	27	39	'	apostrophe
(	28	40	(	left parenthesis
)	29	41	)	right parenthesis
*	2A	42	*	asterisk
+	2B	43	+	plus
,	2C	44	,	comma
-	2D	45	-	hyphen, minus
/	2F	47	/	slash, slant
0	30	48	0	zero
1	31	49	1	one
2	32	50	2	two
3	33	51	3	three
4	34	52	4	four
5	35	53	5	five
6	36	54	6	six
7	37	55	7	seven
8	38	56	8	eight
9	39	57	9	nine
:	3A	58	:	colon
;	3B	59	;	semicolon
<	3C	60	<	less than
=	3D	61	=	equals
>	3E	62	>	greater than
?	3F	63	?	question mark
@	40	64	@	commercial at sign

Char	Hex Value	Decimal Value	ASCII Abbr	Comments
A	41	65	A	uppercase A
B	42	66	B	uppercase B
C	43	67	C	uppercase C
D	44	68	D	uppercase D
E	45	69	E	uppercase E
F	46	70	F	uppercase F
G	47	71	G	uppercase G
H	48	72	H	uppercase H
I	49	73	I	uppercase I
J	4A	74	J	uppercase J
K	4B	75	K	uppercase K
L	4C	76	L	uppercase L
M	4D	77	M	uppercase M
N	4E	78	N	uppercase N
O	4F	79	O	uppercase O
P	50	80	P	uppercase P
Q	51	81	Q	uppercase Q
R	52	82	R	uppercase R
S	53	83	S	uppercase S
T	54	84	T	uppercase T
U	55	85	U	uppercase U
V	56	86	V	uppercase V
W	57	87	W	uppercase W
X	58	88	X	uppercase X
Y	59	89	Y	uppercase Y
Z	5A	90	Z	uppercase Z
[	5B	91	[	left bracket
\	5C	92	\	backslash
]	5D	93	]	right bracket
^	5E	94	^	up-arrow, circumflex, caret
_	5F	95	UND	back-arrow, underscore
`	60	96	GRA	grave accent

Char	Hex Value	Decimal Value	ASCII Abbr	Comments
a	61	97	LCA	lowercase a
b	62	98	LCB	lowercase b
c	63	99	LCC	lowercase c
d	64	100	LCD	lowercase d
e	65	101	LCE	lowercase e
f	66	102	LCF	lowercase f
g	67	103	LCG	lowercase g
h	68	104	LCH	lowercase h
i	69	105	LCI	lowercase i
j	6A	106	LCJ	lowercase j
k	6B	107	LCK	lowercase k
l	6C	108	LCL	lowercase l
m	6D	109	LCM	lowercase m
n	6E	110	LCN	lowercase n
o	6F	111	LCO	lowercase o
p	70	112	LCP	lowercase p
q	71	113	LCQ	lowercase q
r	72	114	LCR	lowercase r
s	73	115	LCS	lowercase s
t	74	116	LCT	lowercase t
u	75	117	LCU	lowercase u
v	76	118	LCV	lowercase v
w	77	119	LCW	lowercase w
x	78	120	LCX	lowercase x
y	79	121	LCY	lowercase y
z	7A	122	LCZ	lowercase z
{	7B	123	LBR	left brace
|	7C	124	VLN	vertical line
}	7D	125	RBR	right brace
~	7E	126	TIL	tilde
	7F	127	DEL,RO	delete, rubout

N IMPLEMENTATION SPECIFICATIONS

This appendix lists F77L's implementation specifications.

N.1 Compiler Limits

The maximum number of array dimensions is seven.

A maximum of 65536 bytes is allowed per subprogram for generated code.

A repeat count in DATA initialization must not exceed 16383.

The length of CHARACTER constants appearing in type statements in initialization context and in DATA statements is limited to 16383.

The maximum size of a COMMON or array is 2**20 bytes.

At most, seven digits of precision may be printed for a REAL number, and sixteen digits for a DOUBLE PRECISION number.

The maximum length of a direct record is 32767 bytes.

The maximum number of records in a direct file is 2,147,483,647.

A source file is limited to 32767 lines.

The maximum number of executable statements in a program unit is 7280.

The maximum nest depth of nested INCLUDE files is limited by the FILES= parameter in your CONFIG.SYS File (see your DOS Manual). DOS uses five "handles" and the compiler can use up to five more with no INCLUDES. Each nested INCLUDE will use one "handle".

The path length specification to the compiler directory is limited to 41 characters.

A format repeat count is limited to 32767.

The width of a numeric format field is limited to 40. The width of a character (A) field is limited to 32767.

The maximum value for TL or TR or for an X format repeat is 128.

The maximum nesting depth of parentheses in FORMAT statements is 10.

A CHARACTER variable or array element may not be over 32768 bytes long.

The compiler uses dynamic storage allocation for all of its internal tables. The amount of total table space available is limited only by the amount of real memory available to the compiler. The following limits are also subject to the real memory limit.

> The maximum number of identifiers per program unit is 4095.
>
> The literal space for identifier names is 32767 bytes per program unit.
>
> The maximum number of arguments that can be passed to a subprogram is at least 400.
>
> The maximum number of DO Loops per program unit is 4095.
>
> The maximum nesting depth for IF...THEN...ELSE structures and DO Loops is 1000.
>
> The maximum number of labels per program unit is 4680.
>
> The number of GO TO statements is unlimited.
>
> The total number of bytes allowed for a continued statement is 65520.
>
> The total number of bytes allowed for all EQUIVALENCE statements in a program unit is 65520.

N.2 Runtime Limits

The maximum unit number is 32767.

A format repeat count is limited to 32767.

The maximum nesting depth of parentheses in FORMAT statements is 10.

Tabbing backwards cannot be used to the right of character position 128 in a record being output to the console or other non-disk device.

The filename length in the file specification of the OPEN or INQUIRE statements is limited to a maximum of 51 characters.

Stack storage is used for passing arguments, temporary storage and allocating local, unsaved initialized data areas. Standard model F77L program units may have approximately 65000 bytes of stack storage per program unit. Fixed-stack model F77L programs may have up to 65536 bytes of stack storage for the entire program (see section A.5.2).

The Random Number Generators RND(), RRAND() and RANDS() have been statistically tested for randomness. The CHI squared values for the tests of uniformity and independence are 67.2 and 40, which are well within the expected range of values. The following table shows the theoretical and actual values for mean, variance, uniformity and independence.

Value Type	Theoretical Value	Actual Value
Mean	0.5	0.501
Variance	0.08333	0.08226
Uniformity	99	67.2
Independence	99	40

N.2.1 Numeric Limits

A REAL datum or either part of a COMPLEX datum has an absolute value range of from 1.18E–38 to 3.40E+38 and a precision of 10E+7 (24 bits).

A REAL*8 (DOUBLE PRECISION) datum or either part of a COMPLEX*16 datum has an absolute value range of from 2.23E–308 to 1.79E+308, and a precision of of 10E16 (53 bits).

An INTEGER*2 has a value range of from –32768 to 32767. An INTEGER*4 has a value range of from –2147483648 to 2147483647. Note that the compiler will not accept –32768 as an INTEGER*2 constant, or –2147483648 as an INTEGER*4 constant.

N.3 F77L FORTRAN Extensions

The following pages list F77L's extensions to the FORTRAN 77 Standard. These extensions are grouped by category for ease of reference.

CHARACTER SET

- & Ampersand
- $ Dollar Sign
- ! Exclamation Point
- > Greater Than Symbol
- < Less Than Symbol
- " Quotation Mark
- _ Underscore

KEYWORDS

- BC EXTERNAL
- CHAIN
- DO WHILE
- END DO
- INCLUDE
- INPUT
- LC EXTERNAL
- MSC EXTERNAL
- MS EXTERNAL
- NAMELIST
- RECURSIVE

STATEMENTS

- BC EXTERNAL
- CHAIN
- DO WHILE
- END DO
- INPUT
- IMPLICIT NONE: forces use of type statements
- INCLUDE (optionally nested)
- LC EXTERNAL
- MSC EXTERNAL
- NAMELIST
- OPTION BREAK

CONTINUATION LINES

The number of continuation lines is limited only by the amount of dynamic storage available to the compiler and can exceed the FORTRAN 77 Standard's 19 continuation line requirement.

F77L's FEATURES & FUNCTIONS

Up to 31 character names including $ and _.

Initialization in type statements.

Incomplete LABEL lists for ARITHMETIC IF statements.

RECURSION: variables can be allocated on the stack and subprograms can call themselves.

Ability to specify lengths of INTEGER, REAL, LOGICAL and COMPLEX data types using *n.

TRAILING COMMENTS

INTEGER*2

LOGICAL*1

COMPLEX*16

< and > symbols instead of .LT. and .GT.

Optional free-format FORTRAN source files.

HEXADECIMAL CONSTANTS: may appear in value lists of DATA statements.

HOLLERITH CONSTANTS: may appear in value lists of DATA statements, in arguments and in relational expressions.

NARGS(): returns the number of arguments passed to the subprogram in which it appears.

QUOTATION MARK ("): may be used to delimit CHARACTER constants. Quotation marks can be substituted for apostrophes.

OPEN STATEMENT EXTENSIONS

ACCESS= "TRANSPARENT" or "APPEND"
CARRIAGE CONTROL=
POSITION=
ACTION=
PAD=
DELIM=
BLOCKSIZE=*
RECL= for FORMATTED SEQUENTIAL files

NOTE: *The default BLOCKSIZE for F77L is 512 bytes.

OTHER I/O EXTENSIONS

Zw EDIT DESCRIPTOR: hexadecimal I/O

COMMAS (,): in numeric input fields terminate format items.

FLEN= (file length), POSITION=, ACTION=, PAD= and DELIM=: used as INQUIRE statement specifiers.

NUMERIC FORMAT: any numeric format may be used with any numeric data type.

INTRINSIC FUNCTION EXTENSIONS

I2ABS	CDABS
I2DIM	CDCOS
I2MAX0	CDEXP
I2MIN0	CDLOG
I2MOD	CDSIN
I2NINT	CDSQRT
I2SIGN	DCMPLX
INT2	DCONJG
INT4	DIMAG

SYSTEM SUBROUTINES

CALL TIME(result)
CALL DATE(result)
CALL INTRUP(intary, ntrup)
CALL EXIT(ilevel)
CALL TIMER(iticks)
CALL IOSTAT_MSG(message)
CALL FLUSH(iunit)
CALL PROMPT(message)
CALL ENSSTK(istack)
CALL ERROR(message)
CALL SYSTEM(comnd): DOS Commands
CALL GETCL(result): get command line
CALL PRECFILL(filchr): fill character for non-significant digits in numeric output fields,
USERNIT & USERTRM

OTHER FUNCTIONS

IAND, IEOR, IOR & NOT
ISHFT
RANDOM NUMBER GENERATORS
SEGMENT
OFFSET
POINTER

MATH CHIP (NDP) EXCEPTION SUBROUTINES

CALL INVALOP(lflag)
CALL OVEFL(lflag)
CALL UNDFL(lflag)
CALL DVCHK(lflag)
CALL UNDER0(lflag)

RESERVED FOR YOUR NOTES

O LAHEY GRAPHICS LIBRARY

This chapter describes the graphics subroutines and functions that are supplied as extensions to the FORTRAN language. These routines and functions provide a basic level of access to the graphics display capability of your computer, and are invoked in the same manner as user-defined subroutines and functions (see Section 10.3).

The graphics routines support the IBM and compatable Monochrome Display Adapter (MDA), Color Graphics Adapter (CGA), Enhanced Graphics Adapter (EGA), Video Graphics Array (VGA) and the Hercules Graphics Card (HGC). They provide the following functions:

Initialization;
Clear graphics screen;
Set drawing color;
Draw lines;
Draw circles;
Set an individual pixel;
Get the value of a pixel;
Fill a bounded area;
Display text; and
Change user coordinates.

O.1 Variable Naming Conventions

This manual uses the FORTRAN 77 Standard's defaults for variable names where all variable names that begin with I, J, K, L, M, and N are INTEGER variables. Variable names beginning with any other characters are REAL (floating point) variables, except where explicitly defined otherwise.

O.2 Coordinate System

Most of the graphics routines have been designed to be functionally identical to the CalComp-compatible plotting libraries used on many mainframe systems. They are best understood by analogy to a plotter moving a pen over the surface of a sheet of paper. Commands may move the pen from where it is now (the CURRENT POSITION) to any other point on the paper. The pen may be DOWN, thus drawing a line or other figure, or UP, which merely changes the current position.

The logical coordinate system used by the graphics routines is independent of your actual hardware configuration. This means that you need not worry about screen resolutions, number of pixels or aspect ratios. You simply specify what you want to draw in terms of points, lines and circles.

By default, the origin (logical coordinate (0.0, 0.0)) is located at the lower-left corner of the display. Positive X coordinates run from left to right, with (11.0, 0.0) being the lower-right corner of the screen. Positive Y coordinates run vertically, with (0.0, 8.5) at the upper-left corner. These coordinates were chosen to match the size of a standard sheet of paper, and are fairly close to the actual dimensions of most monitors. If you wish to use a different system, origin and scaling may be changed with the PLOT and FACTOR subroutines.

O.3 Initialization

Before your program can use any of the other graphics routines, it must first call the initialization routine, PLOTS. This routine saves the current video mode and switches your display into graphics mode. In most cases, PLOTS is capable of detecting the type of graphics adapter you have. By default, the highest available resolution will be used. However, you can explicitly specify the mode if auto-detection fails, or if you want to use a different mode.

Calling PLOT with IPEN = 999 terminates graphics, and restores the original video mode. You should not call PLOTS more than once without an intervening end-of-plot call, as this will destroy

the memory of the original video state. If this happens accidently, or through abnormal program termination, you can return to text mode with the DOS MODE Command.

O.4 Graphics Routines

Following is an alphabetic list of the available graphics routines, their syntax and usage.

O.4.1 CIRCLE Routine

```
CALL CIRCLE (X, Y, RADIUS)
```

Draws a circle centered at position (X, Y). (X, Y) then becomes the new current position. If RADIUS is less than zero, the circle will be filled.

O.4.2 FACTOR Routine

```
CALL FACTOR (FACT)
```

Sets the drawing scale factor. After a call to FACTOR, all subsequent movements are multiplied by FACT. For example, if FACTOR is called with FACT=2.0, then PLOT is called to draw a line one unit long, then the actual length of the line will be two units.

O.4.3 FILL Routine

```
CALL FILL (N, XARRAY, YARRAY)
```

Fills a polygonal area with the current drawing color. N is the number of vertices of the polygon. XARRAY and YARRAY are arrays containing the X and Y coordinates of each vertex.

The polygon to be filled may be either convex or concave, and may even be self-intersecting. However, FILL may fail in some cases where the user attempts to define hollow polygons. Either define the area to be filled in several sections, or fill the voids using color 0.

NOTE: FILL uses the program stack for temporary storage. There is sufficient space for polygons with up to 30 vertices. If a larger number of vertices is used, call subroutine ENSSTK (Section 12.2.2) to increase the minimum amount of stack storage reserved. Six additional bytes over the default minimum of 240 are needed for each vertex over 30. See the Assembly Language Interface (**Appendix J**) for more information.

O.4.4 GETPIX Routine

CALL GETPIX (X, Y, ICOL)

Returns the color value of the pixel closest to position (X, Y).

O.4.5 NEWPEN Routine

CALL NEWPEN (IPEN)

Sets the current drawing color. The actual color that corresponds to any particular value of IPEN, and the number of colors available, depends on the type of graphics adapter you have installed on your system.

NOTE: An IPEN value of 0 turns pixels off. Setting IPEN larger than the maximum color for your system produces undefined results.

O.4.6 PLOT Routine

CALL PLOT (X, Y, IPEN)

Depending on the value of IPEN, this routine may draw straight lines, move the current position, clear the screen or terminate graphics mode.

IPEN = ± 2: Draw absolute. A straight line is drawn from the current position to coordinates (X, Y). (X, Y) becomes the new current position. The color of the line is specified by the most recent call to NEWPEN.

IPEN = ± 3:	Move absolute. (X, Y) becomes the new current position.
IPEN = ± 12:	Draw relative. A line is drawn from the current position (X0, Y0) to (X0+X, Y0+Y), which then becomes the new current position.
IPEN = ± 13:	Move relative. (X0+X, Y0+Y) becomes the new current position.

In all of the above calls, if IPEN is negative, then (X, Y) becomes the origin (0.0, 0.0) for all subsequent plotting.

IPEN = –999, –23 or 23 (these are equivalent):

Clears the screen for the next display.

IPEN = 999:	Terminate graphics mode and return to the video mode that was saved by the last call to PLOTS.

If PLOT is called with any other value of IPEN, the action is undefined.

O.4.7 PLOTS Routine

```
CALL PLOTS (I1, I2, IMODE)
```

This routine puts the display adapter in graphics mode and initializes the graphics package. In this implemention, I1 is a dummy argument and should always be set to zero.

I2 selects either BIOS or direct video output. If I2 = 0, all video I/O will be done through the computer's built-in BIOS routines. This ensures that your programs will work with virtually all standard graphics modes and cards, and may also be necessary if you are using a windowing or multi-tasking environment.

If I2 = 1, the routines write directly to video memory. This is much faster than using BIOS calls. However, it is possible that it may not work correctly with some non-standard video adapters.

NOTE: The BIOS functions are not available on Hercules cards. Direct video is not available in IBM monochrome (640 by 200) mode.

IMODE specifies the graphics mode to be used. If it is zero, the graphics routine will automatically detect the type of display adapter. If it is non-zero, then it must be either one of the standard IBM BIOS graphics modes, or 99 if you have a Hercules Graphics Card.

O.4.8 SETPIX Routine

```
CALL SETPIX (X, Y, ICOL)
```

Sets the pixel closest to position (X, Y) to ICOL.

NOTE: Calling SETPIX does not change the current position.

O.4.9 WHERE Routine

```
CALL WHERE (X, Y, FACT)
```

X and Y, and FACT are set to the current plotting coordinates and scaling factor.

O.5 Text in Graphics Mode

The normal FORTRAN I/O statements do not mix well with graphics output. Depending on the mode, output from a WRITE may scroll the screen, or simply appear as scattered pixels. The following routines provide alternative I/O facilities.

O.5.1 GTEXT Routine

```
CALL GTEXT (LINE, ICOLUMN, STRING)
```

Prints the text in CHARACTER variable STRING at the line and column specified. GTEXT only prints CHARACTER variables. Numeric data may be formatted by writing it to a CHARACTER variable, then outputting the resulting string.

NOTE: To erase strings that have been written with GTEXT, write a string of spaces to the same position.

O.5.2 ISKEY Function

L = ISKEY ()

A LOGICAL function which returns .TRUE. if a key has been pressed. The value of the key may be gotten with IXKEY, or with a normal READ.

O.5.3 IXKEY Function

KEY = IXKEY ()

An INTEGER*2 function which returns a code value for the keypress waiting in the keyboard input buffer. If the key pressed was a normal ASCII character, the code is the ASCII value. Extended key codes (function keys, cursor keys, etc.) return the DOS extended-key code plus 1000.

O.6 The LPLOT COMMON Block

Several useful values may be accessed through the common block LPLOT, which has the following structure:

COMMON /LPLOT/ NXPIX, NYPIX, MCOL, VERSN

INTEGER*2 NXPIX, NYPIX, MCOL

CHARACTER*40 VERSN

They are:

NXPIX - The number of pixels horizontally.

NYPIX - The number of pixels vertically.

MCOL - The maximum available color number.

VERSN - The software version identification.

O.7 BIOS Graphics Modes

Adapter	Mode	Resolution	Colors
CGA	4	320 by 200	4
CGA	5	320 by 200	2
Mono	6	640 by 200	2
EGA	13	320 by 200	16
EGA	14	640 by 200	16
EGA	15	640 by 200	16
EGA	16	640 by 350	16
VGA	17	640 by 480	2
VGA	18	640 by 480	16
VGA	19	320 by 200	256

O.8 Linking in the Graphics Library

FORTRAN programs that include Lahey Graphics Library routines must be linked with the graphics library specified on the OPTLINK Command Line. For example:

```
OPTLINK MAIN,MAIN.EXE,NUL,GRAPH+F77L;
```

Where:

"OPTLINK" is the Lahey OPTLINK Linker;

"MAIN" is the main FORTRAN program;

"MAIN.EXE" is the executable filename;

"NUL" supresses creation on a MAP File;

"GRAPH" is the Lahey Graphics Library; and

"F77L" is the Lahey FORTRAN Library.

NOTE: The Lahey F77L FORTRAN Library must always be the last library specified on the link command line.

P LAHEY IBM VS & DEC VAX INTRINSIC FUNCTIONS

The functions listed in this appendix are extensions to the FORTRAN 77 Standard which are provided in addition to the intrinsic functions found in **Chapter 11**. These functions will allow easy porting of DEC VAX and IBM VS programs to the PC using any Lahey FORTRAN Language System. Follow the instructions on the quick reference card to use these functions.

RESERVED FOR YOUR NOTES

INDEX

INDEX

INDEX

INDEX

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

RESERVED FOR YOUR NOTES

LAHEY CHANGE OF OWNER AND NEW ADDRESS CARD

COMPLETE THIS CARD IF YOU MOVE OR TRANSFER A LAHEY PRODUCT'S OWNERSHIP.

LAHEY CHANGE OF OWNER AND NEW ADDRESS CARD

Please fill out and return this form if you move or transfer the ownership of a Lahey product

Product________________________ Serial Number______________ Version_______

Old Owner Name____

Old Company Name____

Old Address____

City_________________________________ State______________ Zip__________

Country____________________________________ Tel______________________

New Owner Name____

New Company Name____

New Address____

City_________________________________ State______________ Zip__________

Country____________________________________ Tel______________________

I acknowledge that I have destroyed all copies of the Lahey Software in my possession in accordance with the terms of the Lahey Program License Agreement and have transferred my ownership rights to the above named party.

Signature____ Date________________

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED
STATES

BUSINESS REPLY CARD

FIRST CLASS MAIL PERMIT NO. 23 INCLINE VILLAGE, NV

POSTAGE WILL BE PAID BY ADDRESSEE

LAHEY COMPUTER SYSTEMS, INC.
P.O. BOX 6091
INCLINE VILLAGE, NV 89450

IBM VS & DEC VAX INTRINSIC FUNCTIONS

These library functions are not "intrinsically" defined in the compiler and do not have generic functionality. Users must provide the function type definitions in order for the compiler to use the functions. This is done in one of two ways: within the program unit containing the function, or by using an INCLUDE statement to include the VAXIBM.DEF Definitions File in each program unit containing a function from the library. Refer to the two examples below. Both examples print 4.

Example 1 is correct, yet it results in multiple compiler warning messages, e.g. "defined but never used", because the VAXIBM.DEF File includes definitions for all library functions, regardless of whether or not they are used in a program. When including the VAXIBM.DEF File, compile fully debugged programs with /NW to suppress warning messages.

Example 2 illustrates a more concise approach where the programmer declares the IINT Function to be INTEGER*2. In this case the VAXIBM.DEF File was not needed and the compiler generates no additional warning messages as were output by Example 1.

USAGE:

Example 1

```
INCLUDE VAXIBM.DEF
INTEGER*2 I
REAL*4 R
      R = 4.2
      I = IINT (R)
PRINT*,I
END
```

Example 2

```
INTEGER*2 I,IINT
REAL*4 R
      R = 4.2
      I = IINT (R)
PRINT*, I
END
```

PORTING HINT:

To identify where type definitions are needed, link the program without specifying the library. This forces the linker to output "unresolved externals" messages. These linker error messages point to where you need to explicitly define the function types. Once you have defined the types (as shown in Example 2), then link using the library.

LINKING:

To link under F77L or Lahey Personal FORTRAN 77, type: LINK progname,,,vaxibm.lib;
To link under F77L-EM/16, type: UP LEM progname,,,vaxibmem.lib;
To link under F77L-EM/32, type: UP L32 progname,,,vaxibm3.lib;

NOTES FOR TABLES 1 AND 2:

1 [x] is defined as the largest integer whose magnitude does not exceed the magnitude of x and whose sign is the same as that of x. For example [5.7] = 5. and [-5.7] = -5.

2 The argument of SIND, COSD or TAND must be in degrees. The argument is treated as modulo 360.

3 The result of ASIND, DASIND, ACOSD, DACOSD, ATAND, DATAND, ATAN2D or DATAN2D is in degrees.

4 If the value of the first argument of ATAN2D or DATAN2D is positive, the result is positive. When the value of the first argument is zero, the result will be zero if the second argument is positive and 180 degrees if the second argument is negative. If the value of the first argument is negative, the result is negative. If the value of the second argument is zero, the absolute value of the result is 90 degrees. Both arguments must not have the value zero. The range of the result for ATAN2D or DATAN2D is -180 degrees $<$ result $\leq$ 180 degrees.

5 The argument for the cotangent functions may not approach a multiple of π.

6 The bits in a_1 are numbered from right to left, beginning at zero. Thus, $a_2 = 0$ corresponds to the right-most bit of the argument a_1.

7. The BTEST, IBSET and IBCLR functions are not supported as generic functions. Use caution when porting, since only 4-byte results are returned.

TABLE 1 – DEC VAX INTRINSIC FUNCTIONS

Usage Definition	Intrinsic Name	Number of Arguments	Argument Type	Function Type
Type Conversion Functions				
INTEGER to REAL*4	FLOATI	1	INTEGER*2	REAL*4
	FLOATJ	1	INTEGER*4	REAL*4
INTEGER to REAL*8	DFLOTI	1	INTEGER*2	REAL*8
	DFLOTJ	1	INTEGER*4	REAL*8
REAL*4 to INTEGER	IIFIX	1	REAL*4	INTEGER*2
	JIFIX	1	REAL*4	INTEGER*4
Truncation Functions				
IINT(a)	IINT	1	REAL*4	INTEGER*2
See Note 1	JINT	1	REAL*4	INTEGER*4
	IIDINT	1	REAL*8	INTEGER*2
	JIDINT	1	REAL*8	INTEGER*4
Nearest Integer Functions				
[a+.5 * sign(a)]	ININT	1	REAL*4	INTEGER*2
See Note 1	JNINT	1	REAL*4	INTEGER*4
	IIDNNT	1	REAL*8	INTEGER*2
	JIDNNT	1	REAL*8	INTEGER*4
Absolute Value Functions				
|a|	IIABS	1	INTEGER*2	INTEGER*2
	JIABS	1	INTEGER*4	INTEGER*4
Remainder Functions				
a_1-a_2*[a_1/a_2]	IMOD	2	INTEGER*2	INTEGER*2
	JMOD	2	INTEGER*4	INTEGER*4
Transfer of SIGN Functions				
|a_1| Sign a_2	IISIGN	2	INTEGER*2	INTEGER*2
	JISIGN	2	INTEGER*4	INTEGER*4
Positive Difference Functions				
a_1-(min(a_1,a_2))	IIDIM	2	INTEGER*2	INTEGER*2
	JIDIM	2	INTEGER*4	INTEGER*4
Trigonometric Functions				
Sine	SIND	1	REAL*4	REAL*4
DSIND(a)	DSIND	1	REAL*8	REAL*8
See Note 2				
Cosine	COSD	1	REAL*4	REAL*4
DCOSD(a)	DCOSD	1	REAL*8	REAL*8
See Note 2				
Tangent	TAND	1	REAL*4	REAL*4
DTAND(a)	DTAND	1	REAL*8	REAL*8
See Note 2				
Arc Sine	ASIND	1	REAL*4	REAL*4
DASIND(a)	DASIND	1	REAL*8	REAL*8
Arc Cosine	ACOSD	1	REAL*4	REAL*4
DACOSD(a)	DACOSD	1	REAL*8	REAL*8
Arc Tangent	ATAND	1	REAL*4	REAL*4
DATAND(a)	DATAND	1	REAL*8	REAL*8
See Note 3				
Arc Tangent	ATAN2D	2	REAL*4	REAL*4
DATAN2D(a_1,a_2)	DATAN2D	2	REAL*8	REAL*8
See Notes 3 & 4				
Zero-Extend Functions				
16-bit zero-extend function	IZEXT	1	LOGICAL*1	INTEGER*2
16-bit zero-extend function	IZEXT2	1	INTEGER*2	INTEGER*2
32-bit zero-extend function	JZEXT	1	LOGICAL*4	INTEGER*4
32-bit zero-extend function	JZEXT2	1	INTEGER*2	INTEGER*4
32-bit zero-extend function	JZEXT4	1	INTEGER*4	INTEGER*4

TABLE 1 – VAX INTRINSIC FUNCTIONS CONT'D

Usage Definition	Intrinsic Name	Number of Arguments	Argument Type	Function Type
Maximum Functions				
Maximum	IMAX0	≥2	INTEGER*2	INTEGER*2
JMAX0(a_1,a_2,...a_n)	JMAX0	≥2	INTEGER*4	INTEGER*4
	IMAX1	≥2	REAL*4	INTEGER*2
	JMAX1	≥2	REAL*4	INTEGER*4
	AIMAX0	≥2	INTEGER*2	REAL*4
	AJMAX0	≥2	INTEGER*4	REAL*4
Minimum Functions				
Minimum	IMIN0	≥2	INTEGER*2	INTEGER*2
JMIN0(a_1,a_2,..a_n)	JMIN0	≥2	INTEGER*4	INTEGER*4
	IMIN1	≥2	REAL*4	INTEGER*2
	JMIN1	≥2	REAL*4	INTEGER*4
	AIMIN0	≥2	INTEGER*2	REAL*4
	AJMIN0	≥2	INTEGER*4	REAL*4
Bitwise AND Functions				
Performs a logical AND	IIAND	2	INTEGER*2	INTEGER*2
	JIAND	2	INTEGER*4	INTEGER*4
Bitwise OR Functions				
Performs an inclusive OR	IIOR	2	INTEGER*2	INTEGER*2
	JIOR	2	INTEGER*4	INTEGER*4
Bitwise Exclusive OR Functions				
Performs an exclusive OR	IIEOR	2	INTEGER*2	INTEGER*2
	JIEOR	2	INTEGER*4	INTEGER*4
Bitwise Compliment Functions				
Compliments each bit	INOT	1	INTEGER*2	INTEGER*2
	JNOT	1	INTEGER*4	INTEGER*4
Bitwise Shift Functions				
a_1 logically shifted left a_2 bits	IISHFT	2	INTEGER*2	INTEGER*2
	JISHFT	2	INTEGER*4	INTEGER*4
Bit Extraction Functions				
Extracts bits a_2 through	IIBITS	3	INTEGER*2	INTEGER*2
$a_2 + a_3 - 1$ from a_1	JIBITS	3	INTEGER*4	INTEGER*4
Bit Extraction Subroutine				
CALL MVBITS(a_1,a_2,a_3,a_4,a_5) Where a_1 is source, a_2 is source offset, a_3 is length, a_4 is destination and a_5 is destination offset. The values of $a_2 + a_3$ must < 32 and $a_5 + a_3$ must be ≤ 32.	MVBITS	5	INTEGER*4	–
Bit Set Functions				
Returns the value of a_1 with	IIBSET	2	INTEGER*2	INTEGER*2
bit a_2 of a_1 set to 1	JIBSET	2	INTEGER*4	INTEGER*4
See Notes 6 & 7				
Bit Test Functions				
Returns .TRUE. if bit a_2 of	BITEST	2	INTEGER*2	LOGICAL*1
argument a_1 equals 1	BJTEST	2	INTEGER*4	LOGICAL*4
See Notes 6 & 7				
Bit Clear Functions				
Returns the value of a_1 with	IIBCLR	2	INTEGER*2	INTEGER*2
bit a_2 of a_1 set to 0	JIBCLR	2	INTEGER*4	INTEGER*4
See Notes 6 & 7				
Bitwise Circular Shift Functions				
Circularly shifts rightmost a_3	IISHFTC	3	INTEGER*2	INTEGER*2
bits of argument a_1 by a_2 places	JISHFTC	3	INTEGER*4	INTEGER*4

TABLE 2 – IBM VS INTRINSIC FUNCTIONS

Usage Definition	Intrinsic Name	Number of Arguments	Argument Type	Function Type
Type Conversion Functions				
Conversion to real	DREAL	1	COMPLEX*16	REAL*8
Conversion to double	DFLOAT	1	INTEGER*4	REAL*8
Conversion to integer $y = (\text{sign of } x) \cdot n$ where n is the largest integer $\leq \lvert x \rvert$	HFIX	1	REAL*4	INTEGER*2
Cotangent Functions				
$y = \mathrm{COTAN}(x)$ ***See Note 5***	COTAN	1	REAL*4	REAL*4
Range $\lvert x \rvert < (2^{18} \cdot \pi)$ $y = \mathrm{DCOTAN}(x)$ ***See Note 5*** Range $\lvert x \rvert < (2^{18} \cdot \pi)$	DCOTAN	1	REAL*8	REAL*8
Gamma Functions				
$y=\int_0^\infty u^{x-1}e^{-u}du$ Range $x > 2^{-252}$ and $x < 57.5744$	GAMMA	1	REAL*4	REAL*4
$y=\int_0^\infty u^{x-1}e^{-u}du$ Range $x > 2^{-252}$ and $x < 57.5744$	DGAMMA	1	REAL*8	REAL*8
Log Gamma Functions				
$y=\ln\int_0^\infty u^{x-1}e^{-u}du$ Range $x > 0$ and $x < 4.2913 \cdot 10^{73}$	ALGAMA	1	REAL*4	REAL*4
$y=\ln\int_0^\infty u^{x-1}e^{-u}du$ Range $x > 0$ and $x < 4.2913 \cdot 10^{73}$	DLGAMA	1	REAL*8	REAL*8
Error Functions				
$y=\frac{2}{\sqrt{\pi}}\int_0^x e^{-u^2}du$	ERF	1	REAL*4	REAL*4
$y=\frac{2}{\sqrt{\pi}}\int_x^\infty e^{-u^2}du$	ERFC	1	REAL*4	REAL*4
$y=\frac{2}{\sqrt{\pi}}\int_0^x e^{-u^2}$	DERF	1	REAL*8	REAL*8
$y=\frac{2}{\sqrt{\pi}}\int_x^\infty e^{-u^2}du$	DERFC	1	REAL*8	REAL*8
Bit Test Function Returns .TRUE. if bit a_2 of argument a_1 equals 1 ***See Notes 6 & 7***	BTEST	2	INTEGER*4	LOGICAL*4
Bit Set Function Returns the value of a_1 with bit a_2 of a_1 set to 1 ***See Notes 6 & 7***	IBSET	2	INTEGER*4	INTEGER*4
Bit Clear Function Returns the value of a_1 with bit a_2 of a_1 set to 0 ***See Notes 6 & 7***	IBCLR	2	INTEGER*4	INTEGER*4